```
PROJECT FILE LAYOUT
===================
? .
    ? AndroidManifest.xml
? .idea
    ? misc.xml
    ? modules.xml
    ? workspace.xml
    ? inspectionProfiles
        ? profiles_settings.xml
? java
    ? com
        ? example
            ? holodex
                ? MyApp.kt
                ? auth
                    ? AuthRepository.kt
                    ? AuthViewModel.kt
                    ? LoginScreen.kt
                    ? TokenManager.kt
                ? background
                    ? FavoriteChannelSynchronizer.kt
                    ? HistorySynchronizer.kt
                    ? ISynchronizer.kt
                    ? LikesSynchronizer.kt
                    ? M4AExportWorker.kt
                    ? MetadataUpdateWorker.kt
                    ? MetadataWriter.kt
                    ? PlaylistSynchronizer.kt
                    ? StarredPlaylistSynchronizer.kt
                    ? SyncCoordinator.kt
                    ? SyncLogger.kt
                    ? SyncWorker.kt
                ? data
                    ? api
                        ? AuthenticatedMusicdexApiService.kt
                        ? HolodexApiService.kt
                        ? MusicdexApiService.kt
                        ? PlaylistDto.kt
                        ? PlaylistRequestDtos.kt
                    ? cache
                        ? BrowseListCache.kt
                        ? CacheKey.kt
                        ? CachePolicyAndException.kt
                        ? FetcherResult.kt
                        ? SearchListCache.kt
                    ? db
                        ? AppDatabase.kt
                        ? BrowsePageDao.kt
                        ? CachedPageEntities.kt
                        ? Converters.kt
                        ? DiscoveryDao.kt
                        ? DownloadedItemDao.kt
                        ? FavoriteChannelDao.kt
                        ? HistoryDao.kt
```

```
        ? HolodexSongListConverter.kt
        ? LikedItemDao.kt
        ? LocalDao.kt
        ? LocalEntities.kt
        ? ParentVideoMetadataDao.kt
        ? PlaylistDao.kt
        ? SearchPageDao.kt
        ? StarredPlaylistDao.kt
        ? StarredPlaylistEntity.kt
        ? SyncMetadataDao.kt
        ? VideoDao.kt
        ? entities.kt
        ? mappers
            ? SyncMappers.kt
    ? download
        ? DownloadCompletionObserver.kt
        ? DownloadExceptions.kt
        ? LegacyDownloadScanner.kt
    ? model
        ? AudioStreamDetails.kt
        ? ChannelSearchResult.kt
        ? HolodexSong.kt
        ? HolodexVideoItem.kt
        ? PaginatedVideosResponse.kt
        ? VideoSearchRequest.kt
        ? discovery
            ? ChannelDetails.kt
            ? DiscoveryResponse.kt
            ? FullPlaylist.kt
            ? MusicdexSong.kt
            ? PlaylistStub.kt
    ? repository
        ? DownloadRepository.kt
        ? HolodexRepository.kt
        ? LocalRepository.kt
        ? SearchHistoryRepository.kt
        ? UserPreferencesRepository.kt
        ? YouTubeStreamRepository.kt
    ? source
        ? CacheSchemeDataSourceFactory.kt
? di
    ? AppModule.kt
    ? AuthModule.kt
    ? CacheModule.kt
    ? DatabaseModule.kt
    ? NetworkModule.kt
    ? PlaybackModule.kt
    ? Qualifiers.kt
    ? RepositoryModule.kt
    ? SyncModule.kt
    ? UseCaseModule.kt
? export
? extractor
    ? DownloaderImpl.kt
? playback
```

```
? PlaybackRequestManager.kt
? data
    ? mapper
        ? MediaItemMapper.kt
        ? PersistedPlaybackStateMapper.kt
    ? model
        ? PersistedPlaybackItemEntity.kt
        ? PersistedPlaybackStateDao.kt
        ? PersistedPlaybackStateEntity.kt
    ? persistence
        ? PlaybackStatePersistenceManager.kt
    ? preload
        ? PreloadConfiguration.kt
        ? PreloadStatusController.kt
    ? queue
        ? PlaybackQueueManager.kt
        ? PlaybackQueueState.kt
        ? QueueAction.kt
        ? ShuffleOrderProvider.kt
    ? repository
        ? HolodexStreamResolverRepositoryImpl.kt
        ? Media3PlaybackRepositoryImpl.kt
        ? RoomPlaybackStateRepositoryImpl.kt
    ? source
        ? HolodexResolvingDataSource.kt
        ? StreamResolutionCoordinator.kt
    ? tracker
        ? PlaybackProgressTracker.kt
? domain
    ? model
        ? DomainPlaybackProgress.kt
        ? DomainPlaybackState.kt
        ? DomainRepeatMode.kt
        ? DomainShuffleMode.kt
        ? PersistedPlaybackData.kt
        ? PlaybackItem.kt
        ? PlaybackQueue.kt
        ? StreamDetails.kt
    ? repository
        ? PlaybackRepository.kt
        ? PlaybackStateRepository.kt
        ? StreamResolverRepository.kt
    ? usecase
        ? AddItemToQueueUseCase.kt
        ? AddItemsToQueueUseCase.kt
        ? AddOrFetchAndAddUseCase.kt
        ? ClearQueueUseCase.kt
        ? GetPlayerSessionIdUseCase.kt
        ? LoadPlaybackStateUseCase.kt
        ? ObserveCurrentPlayingItemUseCase.kt
        ? ObservePlaybackProgressUseCase.kt
        ? ObservePlaybackQueueUseCase.kt
        ? ObservePlaybackStateUseCase.kt
        ? PausePlaybackUseCase.kt
        ? PlayItemsUseCase.kt
```

```
            ? PlayerControlUseCase.kt
            ? QueueManagementUseCase.kt
            ? ReleasePlaybackResourcesUseCase.kt
            ? RemoveItemFromQueueUseCase.kt
            ? ReorderQueueItemUseCase.kt
            ? ResolveStreamUrlUseCase.kt
            ? ResumePlaybackUseCase.kt
            ? SavePlaybackStateUseCase.kt
            ? SeekPlaybackUseCase.kt
            ? SetRepeatModeUseCase.kt
            ? SetScrubbingUseCase.kt
            ? SetShuffleModeUseCase.kt
            ? SkipToNextItemUseCase.kt
            ? SkipToPreviousItemUseCase.kt
            ? SkipToQueueItemUseCase.kt
    ? player
        ? Media3PlayerController.kt
        ? MediaControllerManager.kt
        ? TimelineSynchronizer.kt
    ? util
        ? PlaybackUtil.kt
        ? PlayerStateMapper.kt
? service
    ? HolodexDownloadService.kt
    ? MediaPlaybackService.kt
? ui
    ? MainActivity.kt
    ? MainScreenScaffold.kt
    ? composables
        ? ApiKeyInputScreen.kt
        ? CarouselShelf.kt
        ? ChannelCard.kt
        ? CustomPagedUnifiedList.kt
        ? FullPlayerScreen.kt
        ? HeroCard.kt
        ? HeroCarousel.kt
        ? ItemOptionsMenu.kt
        ? Media3PlayerControls.kt
        ? MiniPlayerWithProgressBar.kt
        ? PlayerBackground.kt
        ? PlaylistArtwork.kt
        ? PlaylistCard.kt
        ? PlaylistManagementDialogs.kt
        ? StateDisplayComposables.kt
        ? UnifiedGridItem.kt
        ? UnifiedListItem.kt
        ? sheets
            ? BrowseFiltersSheet.kt
    ? dialogs
        ? AddExternalChannelDialog.kt
        ? CreatePlaylistDialog.kt
        ? SelectPlaylistDialog.kt
    ? screens
        ? ChannelScreen.kt
        ? DiscoveryScreen.kt
```

```
        ? DownloadsScreen.kt
        ? EditablePlaylistHeader.kt
        ? ExternalChannelScreen.kt
        ? FavoritesScreen.kt
        ? ForYouScreen.kt
        ? FullListViewScreen.kt
        ? HistoryScreen.kt
        ? HomeScreen.kt
        ? LibraryScreen.kt
        ? PlaylistDetailsScreen.kt
        ? PlaylistsScreen.kt
        ? SettingsScreen.kt
        ? VideoDetailsScreen.kt
        ? navigation
            ? AppDestinations.kt
            ? BottomNavItem.kt
            ? HolodexNavHost.kt
    ? theme
        ? Color.kt
        ? Shape.kt
        ? Theme.kt
        ? Type.kt
? util
    ? ArtworkResolver.kt
    ? ComposableUtils.kt
    ? Extract_util.kt
    ? ImageUtils.kt
    ? PaletteExtractor.kt
    ? PlaylistFormatter.kt
    ? VideoFilteringUtil.kt
? viewmodel
    ? ChannelDetailsViewModel.kt
    ? DiscoveryViewModel.kt
    ? DownloadsViewModel.kt
    ? ExternalChannelViewModel.kt
    ? FavoritesViewModel.kt
    ? FullListViewModel.kt
    ? FullPlayerViewModel.kt
    ? HistoryViewModel.kt
    ? PlaybackUiStateSelectors.kt
    ? PlaybackViewModel.kt
    ? PlaylistDetailsViewModel.kt
    ? PlaylistManagementViewModel.kt
    ? SettingsViewModel.kt
    ? SharedViewModelTypes.kt
    ? UnifiedDisplayItem.kt
    ? VideoDetailsViewModel.kt
    ? VideoListViewModel.kt
    ? autoplay
        ? AutoplayItemProvider.kt
        ? ContinuationManager.kt
    ? mappers
        ? UnifiedDisplayItemMapper.kt
    ? state
        ? BrowseFilterState.kt
```

```
                        ? UiState.kt
? res
    ? drawable
        ? ic_default_album_art_placeholder.xml
        ? ic_error_image.xml
        ? ic_launcher_background.xml
        ? ic_launcher_foreground.xml
        ? ic_like_empty.xml
        ? ic_notification_small.xml
        ? ic_pause.xml
        ? ic_placeholder_image.xml
        ? ic_play_arrow.xml
        ? ic_repeat_off_24.xml
        ? ic_repeat_on_24.xml
        ? ic_repeat_one_24.xml
        ? ic_shuffle_off_24.xml
        ? ic_shuffle_on_24.xml
        ? ic_skip_next.xml
        ? ic_skip_previous.xml
        ? ic_stat_music_note.xml
        ? ic_twitter.xml
        ? ic_youtube.xml
        ? twitter.xml
        ? youtube.xml
    ? layout
    ? mipmap-anydpi-v26
        ? ic_launcher.xml
    ? mipmap-hdpi
    ? mipmap-mdpi
    ? mipmap-xhdpi
    ? mipmap-xxhdpi
    ? mipmap-xxxhdpi
    ? values
        ? Theme.xml
        ? attrs.xml
        ? colors.xml
        ? strings.xml
    ? values-night
        ? themes.xml
    ? xml
        ? backup_rules.xml
        ? data_extraction_rules.xml

==================
CODE CONTENT

// File: AndroidManifest.xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    package="com.example.holodex">

    <!-- Permissions -->
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.WAKE_LOCK" />
```

```
<uses-permission android:name="android.permission.FOREGROUND_SERVICE" />
<uses-permission android:name="android.permission.POST_NOTIFICATIONS"/>
<uses-permission android:name="android.permission.FOREGROUND_SERVICE_MEDIA_PLAYBACK" />
<uses-permission android:name="android.permission.FOREGROUND_SERVICE_DATA_SYNC" />

<!-- START OF FIX: Add permissions for reading media -->
<!-- For Android 13 (API 33) and above -->
<uses-permission android:name="android.permission.READ_MEDIA_AUDIO" />

<!-- For Android 12 (API 32) and below.
     The maxSdkVersion ensures this is only requested on older devices
     where it's necessary for MediaStore to scan all audio files. -->
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"
    android:maxSdkVersion="32" />
<!-- END OF FIX -->

<!-- We keep the old write permission for legacy devices where it might be needed -->
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"
    android:maxSdkVersion="28" />


<application
    android:name=".MyApp"
    android:allowBackup="true"
    android:dataExtractionRules="@xml/data_extraction_rules"
    android:fullBackupContent="@xml/backup_rules"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/Theme.Holodex"
    android:usesCleartextTraffic="true"
    tools:targetApi="34"
    android:enableOnBackInvokedCallback="true">

    <!-- (The rest of the file is unchanged) -->

    <provider
        android:name="androidx.startup.InitializationProvider"
        android:authorities="${applicationId}.androidx-startup"
        android:exported="false"
        tools:node="merge">

        <meta-data
            android:name="androidx.work.WorkManagerInitializer"
            android:value="androidx.startup"
            tools:node="remove" />
    </provider>
    <activity
        android:name=".ui.MainActivity"
        android:exported="true"
        android:windowSoftInputMode="adjustResize"
        android:launchMode="singleTop">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
```

```
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <service
            android:name=".service.MediaPlaybackService"
            android:foregroundServiceType="mediaPlayback"
            android:exported="true">
            <intent-filter>
                <action android:name="androidx.media3.session.MediaSessionService"/>
            </intent-filter>
        </service>

        <service
            android:name=".service.HolodexDownloadService"
            android:exported="false"
            android:foregroundServiceType="dataSync">
            <intent-filter>
                <action android:name="androidx.media3.exoplayer.download.DownloadService"/>
            </intent-filter>
        </service>

    </application>

</manifest>

// File: .idea\misc.xml
<?xml version="1.0" encoding="UTF-8"?>
<project version="4">
  <component name="ProjectRootManager" version="2" project-jdk-name="Python 3.11" project-jdk-
</project>

// File: .idea\modules.xml
<?xml version="1.0" encoding="UTF-8"?>
<project version="4">
  <component name="ProjectModuleManager">
    <modules>
      <module fileurl="file://$PROJECT_DIR$/.idea/main.iml" filepath="$PROJECT_DIR$/.idea/main
    </modules>
  </component>
</project>

// File: .idea\workspace.xml
<?xml version="1.0" encoding="UTF-8"?>
<project version="4">
  <component name="ChangeListManager">
    <list default="true" id="071e5753-0ef0-489e-9c83-db4f6a13ebe4" name="Changes" comment="" /
    <option name="SHOW_DIALOG" value="false" />
    <option name="HIGHLIGHT_CONFLICTS" value="true" />
    <option name="HIGHLIGHT_NON_ACTIVE_CHANGELIST" value="false" />
    <option name="LAST_RESOLUTION" value="IGNORE" />
  </component>
  <component name="ProjectColorInfo"><![CDATA[{
  "associatedIndex": 8
}]]></component>
  <component name="ProjectId" id="2y3tTufEtGAMlv9Qc9BPajVvJjG" />
```

```
  <component name="ProjectViewState">
    <option name="hideEmptyMiddlePackages" value="true" />
    <option name="showLibraryContents" value="true" />
  </component>
  <component name="PropertiesComponent"><![CDATA[{
  "keyToString": {
    "ModuleVcsDetector.initialDetectionPerformed": "true",
    "RunOnceActivity.ShowReadmeOnStart": "true",
    "ignore.virus.scanning.warn.message": "true"
  }
}]]></component>
  <component name="SharedIndexes">
    <attachedChunks>
      <set>
        <option value="bundled-python-sdk-348a24fa61fa-5312c7369657-com.jetbrains.pycharm.comm
      </set>
    </attachedChunks>
  </component>
  <component name="TaskManager">
    <task active="true" id="Default" summary="Default task">
      <changelist id="071e5753-0ef0-489e-9c83-db4f6a13ebe4" name="Changes" comment="" />
      <created>1749075097916</created>
      <option name="number" value="Default" />
      <option name="presentableId" value="Default" />
      <updated>1749075097916</updated>
    </task>
    <servers />
  </component>
</project>

// File: .idea\inspectionProfiles\profiles_settings.xml
<component name="InspectionProjectProfileManager">
  <settings>
    <option name="USE_PROJECT_PROFILE" value="false" />
    <version value="1.0" />
  </settings>
</component>

// File: java\com\example\holodex\MyApp.kt
package com.example.holodex

import android.app.Application
import android.app.NotificationChannel
import android.app.NotificationManager
import android.content.Intent
import android.util.Log
import androidx.core.net.toUri
import androidx.hilt.work.HiltWorkerFactory
import androidx.lifecycle.DefaultLifecycleObserver
import androidx.lifecycle.LifecycleOwner
import androidx.lifecycle.ProcessLifecycleOwner
import androidx.media3.common.util.UnstableApi
import androidx.work.Configuration
import coil.ImageLoader
import coil.ImageLoaderFactory
```

```
import coil.annotation.ExperimentalCoilApi
import coil.disk.DiskCache
import coil.memory.MemoryCache
import coil.util.DebugLogger
import com.example.holodex.data.db.DownloadStatus
import com.example.holodex.data.db.DownloadedItemDao
import com.example.holodex.data.download.DownloadCompletionObserver
import com.example.holodex.data.repository.DownloadRepository
import com.example.holodex.data.repository.HolodexRepository
import com.example.holodex.di.ApplicationScope
import com.example.holodex.extractor.DownloaderImpl
import dagger.hilt.android.HiltAndroidApp
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.delay
import kotlinx.coroutines.flow.first
import kotlinx.coroutines.launch
import kotlinx.coroutines.withContext
import okhttp3.OkHttpClient
import org.schabi.newpipe.extractor.NewPipe
import org.schabi.newpipe.extractor.localization.ContentCountry
import org.schabi.newpipe.extractor.localization.Localization
import timber.log.Timber
import java.io.File
import java.util.Locale
import java.util.concurrent.TimeUnit
import javax.inject.Inject
import kotlin.system.exitProcess

@UnstableApi
@HiltAndroidApp
// --- FIX: Implement Configuration.Provider ---
class MyApp : Application(), ImageLoaderFactory, DefaultLifecycleObserver, Configuration.Provi

    @Inject
    lateinit var workerFactory: HiltWorkerFactory

    @Inject
    lateinit var holodexRepository: HolodexRepository

    @Inject
    lateinit var downloadRepository: DownloadRepository

    @Inject
    lateinit var downloadManager: androidx.media3.exoplayer.offline.DownloadManager

    @Inject
    lateinit var downloadedItemDao: DownloadedItemDao

    @Inject
    lateinit var imageLoader: ImageLoader

    @Inject
    lateinit var downloadCompletionObserver: DownloadCompletionObserver
```

```
@Inject
@ApplicationScope
lateinit var appScope: CoroutineScope

override val workManagerConfiguration: Configuration
    get() = Configuration.Builder()
        .setWorkerFactory(workerFactory)
        .setMinimumLoggingLevel(if (BuildConfig.DEBUG) Log.DEBUG else Log.INFO)
        .build()



override fun onCreate() {
    super<Application>.onCreate()


    Timber.i("? WorkManager has been explicitly initialized with HiltWorkerFactory.")

    // The rest of the startup sequence can now proceed safely.
    ProcessLifecycleOwner.get().lifecycle.addObserver(this)

    if (BuildConfig.DEBUG) {
        Timber.plant(Timber.DebugTree())
    }

    createNotificationChannels()

    Timber.d("Initializing DownloadCompletionObserver...")
    downloadCompletionObserver.initialize()
    Timber.d("DownloadCompletionObserver initialized.")

    appScope.launch {
        holodexRepository.cleanupExpiredCacheEntries()
    }

    val downloaderOkHttpClient = OkHttpClient.Builder()
        .connectTimeout(60, TimeUnit.SECONDS)
        .readTimeout(60, TimeUnit.SECONDS)
        .build()

    NewPipe.init(
        DownloaderImpl(downloaderOkHttpClient),
        Localization.fromLocale(Locale.JAPAN),
        ContentCountry(Locale.JAPAN.country)
    )
}

// The rest of your MyApp.kt file is unchanged.
// The following methods are included for completeness but require no changes.
@UnstableApi
override fun onStart(owner: LifecycleOwner) {
    Timber.i("MyApp entering foreground (onStart lifecycle event). Triggering all reconcil
    appScope.launch {
        reconcileActiveDownloadStates()
```

```
        downloadRepository.rescanStorageForDownloads()
        downloadRepository.reconcileAllDownloads()
    }
}


@UnstableApi
private fun reconcileActiveDownloadStates() {
    appScope.launch {
        try {
            Timber.d("MyApp Reconcile: Starting ACTIVE download state reconciliation.")

            val appDbDownloads = downloadedItemDao.getAllDownloads().first()
            val media3ActiveDownloads = downloadManager.currentDownloads
            val media3ActiveDownloadIds = media3ActiveDownloads.map { it.request.id }.toSe

            for (appDbItem in appDbDownloads) {
                if (appDbItem.downloadStatus == DownloadStatus.DOWNLOADING || appDbItem.do
                    if (!media3ActiveDownloadIds.contains(appDbItem.videoId)) {
                        Timber.w("MyApp Reconcile: Item ${appDbItem.videoId} is stuck in a
                        downloadedItemDao.updateStatus(appDbItem.videoId, DownloadStatus.F
                    } else {
                        val media3Download =
                            media3ActiveDownloads.find { it.request.id == appDbItem.videoI

                        if (media3Download?.state == androidx.media3.exoplayer.offline.Dow
                            Timber.w("MyApp Reconcile: Item ${appDbItem.videoId} is FAILED
                            downloadedItemDao.updateStatus(
                                appDbItem.videoId,
                                DownloadStatus.FAILED
                            )
                        }
                    }
                }
            }
            Timber.d("MyApp Reconcile: Active download state reconciliation finished.")
        } catch (e: Exception) {
            Timber.e(e, "MyApp Reconcile: Error during ACTIVE download state reconciliatio
        }
    }
}


private fun createNotificationChannels() {
    val notificationManager = getSystemService(NOTIFICATION_SERVICE) as NotificationManage

    val downloadChannel = NotificationChannel(
        "download_channel",
        getString(R.string.download_notification_channel_name),
        NotificationManager.IMPORTANCE_LOW
    ).apply {
        description =
            getString(R.string.download_notification_channel_description)
    }
```

```
    val playbackChannel = NotificationChannel(
        "holodex_playback_channel_v3",
        getString(R.string.playback_notification_channel_name),
        NotificationManager.IMPORTANCE_LOW
    ).apply {
        description = getString(R.string.playback_notification_channel_description)
    }

    notificationManager.createNotificationChannel(downloadChannel)
    notificationManager.createNotificationChannel(playbackChannel)
    Timber.d("Notification channels created: 'download_channel' and 'holodex_playback_chan
}


override fun newImageLoader(): ImageLoader {
    return ImageLoader.Builder(this)
        .memoryCache {
            MemoryCache.Builder(this)
                .maxSizePercent(0.25)
                .build()
        }
        .diskCache {
            DiskCache.Builder()
                .directory(this.cacheDir.resolve("image_cache_v1"))
                .maxSizeBytes(50L * 1024L * 1024L) // 50MB
                .build()
        }
        .okHttpClient {
            OkHttpClient.Builder()
                .build()
        }
        .respectCacheHeaders(false)
        .apply {
            if (BuildConfig.DEBUG) {
                logger(DebugLogger())
            }
        }
        .build()
}


internal fun reconcileCompletedDownloads() {
    appScope.launch {
        try {
            Timber.d("MyApp Reconcile: Verifying file existence for completed downloads...
            val completedDownloads = downloadedItemDao.getAllDownloads()
                .first()
                .filter { it.downloadStatus == DownloadStatus.COMPLETED }

            for (item in completedDownloads) {
                var fileExists = false
                val uriString = item.localFileUri

                if (!uriString.isNullOrBlank()) {
                    try {
```

```
                        contentResolver.openInputStream(uriString.toUri())?.use {
                            fileExists = true
                        }
                    } catch (_: Exception) {
                        fileExists = false
                    }
                }

                if (!fileExists) {
                    Timber.w("MyApp Reconcile: File for item ${item.videoId} is missing. T
                    downloadRepository.deleteDownloadById(item.videoId)
                }
            }
        } catch (e: Exception) {
            Timber.e(
                e,
                "MyApp Reconcile: CRITICAL error during completed download reconciliation.
            )
        }
    }
}


@OptIn(ExperimentalCoilApi::class)
@UnstableApi
fun clearAllAppCachesOnDemand(callback: (Boolean) -> Unit) {
    appScope.launch {
        var allSuccess = true

        try {
            withContext(Dispatchers.IO) {
                val mediaCacheDir = File(applicationContext.cacheDir, "exoplayer_media_cac
                val downloadCacheDir =
                    File(applicationContext.getExternalFilesDir(null), "downloads")

                if (mediaCacheDir.exists()) mediaCacheDir.deleteRecursively()
                if (downloadCacheDir.exists()) downloadCacheDir.deleteRecursively()

                Timber.i("Force-deleted ExoPlayer cache directories.")
            }
        } catch (e: Exception) {
            Timber.e(e, "Error during manual deletion of ExoPlayer caches.")
            allSuccess = false
        }

        try {
            imageLoader.diskCache?.clear()
            imageLoader.memoryCache?.clear()
            Timber.i("Coil image caches cleared.")
        } catch (e: Exception) {
            Timber.e(e, "Error clearing Coil image cache.")
            allSuccess = false
        }

        try {
```

```
                holodexRepository.clearAllCachedData()
                Timber.i("Holodex repository data cleared.")
            } catch (e: Exception) {
                Timber.e(e, "Error clearing Holodex repository data.")
                allSuccess = false
            }

            Timber.i("Application caches clear attempt finished. Success: $allSuccess")
            callback(allSuccess)

            withContext(Dispatchers.Main) {
                delay(500)
                val packageManager = applicationContext.packageManager
                val intent =
                    packageManager.getLaunchIntentForPackage(applicationContext.packageName)
                val componentName = intent!!.component
                val mainIntent = Intent.makeRestartActivityTask(componentName)
                applicationContext.startActivity(mainIntent)
                exitProcess(0)
            }
        }
    }
}

// File: java\com\example\holodex\auth\AuthRepository.kt
// File: java/com/example/holodex/auth/AuthRepository.kt
// (Create this new file)

package com.example.holodex.auth

import com.example.holodex.data.api.HolodexApiService
import com.example.holodex.data.api.LoginRequest
import net.openid.appauth.AuthorizationService
import net.openid.appauth.TokenRequest
import net.openid.appauth.TokenResponse
import kotlin.coroutines.suspendCoroutine
import kotlin.coroutines.resume
import kotlin.coroutines.resumeWithException

/**
 * Orchestrates the entire authentication flow, from exchanging the Discord
 * auth code to logging into the Holodex backend.
 */
class AuthRepository(
    private val holodexApiService: HolodexApiService,
    private val authService: AuthorizationService
) {

    /**
     * Exchanges a one-time authorization code from Discord for an access token.
     * This is a suspending function that wraps the AppAuth callback-based API.
     */
    suspend fun exchangeDiscordCodeForToken(tokenRequest: TokenRequest): TokenResponse {
        return suspendCoroutine { continuation ->
            authService.performTokenRequest(tokenRequest) { response, ex ->
```

```
                if (response != null) {
                    continuation.resume(response)
                } else {
                    continuation.resumeWithException(ex ?: IllegalStateException("Token exchan
                }
            }
        }
    }

    /**
     * Uses the Discord access token to log into the Holodex backend and get a JWT.
     */
    suspend fun loginToHolodex(discordAccessToken: String): String {
        val request = LoginRequest(service = "discord", token = discordAccessToken)
        val response = holodexApiService.login(request)

        if (response.isSuccessful && response.body() != null) {
            return response.body()!!.jwt
        } else {
            throw Exception("Holodex login failed: ${response.errorBody()?.string()}")
        }
    }
}

// File: java\com\example\holodex\auth\AuthViewModel.kt
// File: java/com/example/holodex/auth/AuthViewModel.kt
// (Create this new file)

package com.example.holodex.auth

import android.content.Context
import android.content.Intent
import android.util.Base64
import androidx.core.net.toUri
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.example.holodex.BuildConfig
import dagger.hilt.android.lifecycle.HiltViewModel
import dagger.hilt.android.qualifiers.ApplicationContext
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.launch
import net.openid.appauth.AuthorizationRequest
import net.openid.appauth.AuthorizationService
import net.openid.appauth.AuthorizationServiceConfiguration
import net.openid.appauth.ResponseTypeValues
import org.json.JSONObject
import timber.log.Timber
import java.nio.charset.StandardCharsets
import javax.inject.Inject

// Represents the different states of the authentication flow for the UI to observe.
sealed class AuthState {
    object LoggedOut : AuthState()
```

```
    object InProgress : AuthState()
    object LoggedIn : AuthState()
    data class Error(val message: String) : AuthState()
}
private fun getUserIdFromJwt(jwt: String): String? {
    return try {
        val parts = jwt.split(".")
        if (parts.size < 2) return null
        val payload = parts[1]
        val decodedBytes = Base64.decode(payload, Base64.URL_SAFE)
        val decodedString = String(decodedBytes, StandardCharsets.UTF_8)
        val json = JSONObject(decodedString)
        json.optInt("i", -1).takeIf { it != -1 }?.toString()
    } catch (e: Exception) {
        Timber.e(e, "Failed to decode JWT payload")
        null
    }
}
@HiltViewModel
class AuthViewModel @Inject constructor(
    @ApplicationContext private val appContext: Context,
    private val authRepository: AuthRepository,
    private val tokenManager: TokenManager
) : ViewModel() {

    private val _authState = MutableStateFlow<AuthState>(AuthState.LoggedOut)
    val authState: StateFlow<AuthState> = _authState.asStateFlow()

    private val authService = AuthorizationService(appContext)

    // Define the endpoints for Discord's OAuth2 service
    private val serviceConfig = AuthorizationServiceConfiguration(
        "https://discord.com/api/oauth2/authorize".toUri(),
        "https://discord.com/api/oauth2/token".toUri()
    )

    init {
        // On ViewModel creation, check if we are already logged in
        if (tokenManager.getJwt() != null) {
            _authState.value = AuthState.LoggedIn
        }
    }

    /**
     * Creates an intent to launch the Discord login flow in a Custom Tab.
     * This is called by the UI when the user taps the "Login" button.
     */
    fun getAuthorizationRequestIntent(): Intent {
        val authRequest = AuthorizationRequest.Builder(
            serviceConfig,
            BuildConfig.DISCORD_CLIENT_ID,
            ResponseTypeValues.CODE,
            BuildConfig.DISCORD_REDIRECT_URI.toUri()
        )
            .setScope("identify email")
```

```
        // AppAuth automatically generates and includes the PKCE parameters
        .build()

    Timber.d("Created authorization request for Discord.")
    return authService.getAuthorizationRequestIntent(authRequest)
}

/**
 * Handles the redirect intent received from the Custom Tab after the user
 * authorizes the app on Discord.
 */
fun onAuthorizationResponse(intent: Intent) {
    _authState.value = AuthState.InProgress

    val resp = net.openid.appauth.AuthorizationResponse.fromIntent(intent)
    val ex = net.openid.appauth.AuthorizationException.fromIntent(intent)

    if (resp == null) {
        val errorMessage = "Authorization failed: ${ex?.errorDescription ?: "Unknown error
        Timber.e(ex, errorMessage)
        _authState.value = AuthState.Error(errorMessage)
        return
    }

    // The authorization was successful, now exchange the code for a token.
    // AppAuth automatically includes the PKCE code_verifier it generated earlier.
    viewModelScope.launch {
        try {
            Timber.d("Exchanging authorization code for Discord token...")
            val tokenResponse = authRepository.exchangeDiscordCodeForToken(resp.createToke

            val discordAccessToken = tokenResponse.accessToken
            if (discordAccessToken == null) {
                throw IllegalStateException("Discord access token was null")
            }

            Timber.d("Successfully received Discord access token. Now logging into Holodex
            val holodexJwt = authRepository.loginToHolodex(discordAccessToken)
            tokenManager.saveJwt(holodexJwt)
            val userId = getUserIdFromJwt(holodexJwt)
            if (userId != null) {
                tokenManager.saveUserId(userId)
                Timber.i("Successfully logged in, saved Holodex JWT, and extracted User ID
            } else {
                Timber.w("Successfully logged in but could not extract User ID from JWT.")
            }
            _authState.value = AuthState.LoggedIn
            Timber.i("Successfully logged in and saved Holodex JWT.")

        } catch (e: Exception) {
            val errorMessage = "Full login flow failed: ${e.message}"
            Timber.e(e, errorMessage)
            _authState.value = AuthState.Error(errorMessage)
        }
    }
```

```
    }

    fun logout() {
        tokenManager.clearJwt()
        _authState.value = AuthState.LoggedOut
        Timber.i("User logged out and JWT cleared.")
    }

    override fun onCleared() {
        super.onCleared()
        authService.dispose()
    }
}

// File: java\com\example\holodex\auth\LoginScreen.kt
// File: java/com/example/holodex/auth/LoginScreen.kt
// (Create this new file)

package com.example.holodex.auth

import android.widget.Toast
import androidx.activity.compose.rememberLauncherForActivityResult
import androidx.activity.result.contract.ActivityResultContracts
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.Button
import androidx.compose.material3.CircularProgressIndicator
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.runtime.getValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.text.style.TextAlign
import androidx.compose.ui.unit.dp
import androidx.hilt.navigation.compose.hiltViewModel
import androidx.lifecycle.compose.collectAsStateWithLifecycle
import com.example.holodex.R

@Composable
fun LoginScreen(
    authViewModel: AuthViewModel = hiltViewModel(),
    onLoginSuccess: () -> Unit
) {
    val authState by authViewModel.authState.collectAsStateWithLifecycle()
    val context = LocalContext.current
```

```
// This launcher will start the Custom Tab for Discord login.
val authLauncher = rememberLauncherForActivityResult(
    contract = ActivityResultContracts.StartActivityForResult()
) { result ->
    // After the user returns from the Custom Tab, the result intent is passed here.
    result.data?.let { intent ->
        authViewModel.onAuthorizationResponse(intent)
    }
}

// Observe the auth state to react to changes.
LaunchedEffect(authState) {
    when (val state = authState) {
        is AuthState.LoggedIn -> {
            Toast.makeText(context, "Login Successful!", Toast.LENGTH_SHORT).show()
            onLoginSuccess()
        }
        is AuthState.Error -> {
            Toast.makeText(context, "Login Failed: ${state.message}", Toast.LENGTH_LONG).s
        }
        else -> {
            // InProgress or LoggedOut, no side-effect needed here.
        }
    }
}

Box(
    modifier = Modifier
        .fillMaxSize()
        .padding(16.dp),
    contentAlignment = Alignment.Center
) {
    when (authState) {
        is AuthState.InProgress -> {
            CircularProgressIndicator()
        }
        else -> {
            Column(
                horizontalAlignment = Alignment.CenterHorizontally,
                verticalArrangement = Arrangement.Center
            ) {
                Text(
                    text = "Login Required",
                    style = MaterialTheme.typography.headlineSmall,
                )
                Spacer(modifier = Modifier.height(8.dp))
                Text(
                    text = "Please log in with Discord to enable synchronization and other
                    style = MaterialTheme.typography.bodyMedium,
                    textAlign = TextAlign.Center
                )
                Spacer(modifier = Modifier.height(24.dp))
                Button(
                    onClick = {
                        // Launch the authorization intent created by the ViewModel
```

```
                            authLauncher.launch(authViewModel.getAuthorizationRequestIntent())
                        }
                    ) {
                        Text(stringResource(R.string.login_with_discord)) // <-- Add this stri
                    }
                }
            }
        }
    }
}

// File: java\com\example\holodex\auth\TokenManager.kt
// File: java/com/example/holodex/auth/TokenManager.kt

package com.example.holodex.auth

import android.content.Context
import androidx.core.content.edit
import androidx.security.crypto.EncryptedSharedPreferences
import androidx.security.crypto.MasterKeys

class TokenManager(context: Context) {

    companion object {
        private const val PREF_FILE_NAME = "auth_token_prefs"
        private const val KEY_HOLODEX_JWT = "holodex_jwt"
        // --- ADDITION: Key for storing the User ID ---
        private const val KEY_USER_ID = "user_id"
    }

    private val masterKeyAlias = MasterKeys.getOrCreate(MasterKeys.AES256_GCM_SPEC)

    private val sharedPreferences = EncryptedSharedPreferences.create(
        PREF_FILE_NAME,
        masterKeyAlias,
        context,
        EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256_SIV,
        EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256_GCM
    )

    fun saveJwt(token: String) {
        sharedPreferences.edit {
            putString(KEY_HOLODEX_JWT, token)
        }
    }

    fun getJwt(): String? {
        return sharedPreferences.getString(KEY_HOLODEX_JWT, null)
    }

    // --- ADDITION: New methods for User ID ---
    fun saveUserId(id: String) {
        sharedPreferences.edit {
            putString(KEY_USER_ID, id)
        }
```

```
    }

    fun getUserId(): String? {
        return sharedPreferences.getString(KEY_USER_ID, null)
    }
    // --- END OF ADDITION ---

    fun clearJwt() {
        sharedPreferences.edit {
            remove(KEY_HOLODEX_JWT)
            // --- ADDITION: Clear the User ID on logout ---
            remove(KEY_USER_ID)
        }
    }
}

// File: java\com\example\holodex\background\FavoriteChannelSynchronizer.kt
// File: java/com/example/holodex/background/FavoriteChannelSynchronizer.kt (NEW FILE)
package com.example.holodex.background

import com.example.holodex.data.db.SyncStatus
import com.example.holodex.data.repository.HolodexRepository
import javax.inject.Inject

class FavoriteChannelSynchronizer @Inject constructor(
    private val repository: HolodexRepository,
    private val logger: SyncLogger
) : ISynchronizer {
    override val name: String = "FAVORITE_CHANNELS"

    override suspend fun synchronize(): Boolean {
        logger.startSection(name)
        try {
            // --- PHASE 1: UPSTREAM ---
            logger.info("Phase 1: Pushing local changes to server...")
            repository.performUpstreamFavoriteChannelsSync(logger)
            logger.info("Phase 1 complete.")

            // --- PHASE 2: FETCH ---
            logger.info("Phase 2: Fetching remote and local states...")
            val remoteFavs = repository.getRemoteFavoriteChannels()
            val localFavs = repository.getLocalFavoriteChannels()
            logger.info("  -> Fetched ${remoteFavs.size} remote favorites and ${localFavs.size

            // --- PHASE 3: MERGE & RECONCILE ---
            logger.info("Phase 3: Reconciling favorite channels...")
            val remoteMap = remoteFavs.associateBy { it.id }
            val localMap = localFavs.associateBy { it.id }

            // Find new favorites from server to insert locally
            val newFromServer = remoteFavs.filter { !localMap.containsKey(it.id) }
            if (newFromServer.isNotEmpty()) {
                logger.info("  Found ${newFromServer.size} new favorites from server:")
                newFromServer.forEach { p -> logger.logItemAction(LogAction.DOWNSTREAM_INSERT_
                repository.insertNewSyncedFavoriteChannels(newFromServer)
```

```
            }

            // Find favorites deleted on server to delete locally
            val deletedOnServer = localFavs.filter { it.syncStatus == SyncStatus.SYNCED && !re
            if (deletedOnServer.isNotEmpty()) {
                logger.info("  Found ${deletedOnServer.size} favorites deleted on server:")
                deletedOnServer.forEach { p -> logger.logItemAction(LogAction.DOWNSTREAM_DELET
                repository.deleteLocalFavoriteChannels(deletedOnServer.map { it.id })
            }

            logger.info("Phase 3 complete.")
            logger.endSection(name, success = true)
            return true
        } catch (e: Exception) {
            logger.error(e, "Favorite Channels sync failed catastrophically.")
            logger.endSection(name, success = false)
            return false
        }
    }
}

// File: java\com\example\holodex\background\HistorySynchronizer.kt
// File: java/com/example/holodex/background/HistorySynchronizer.kt

package com.example.holodex.background

import com.example.holodex.auth.TokenManager
import com.example.holodex.data.db.HistoryDao
import com.example.holodex.data.db.SyncMetadataDao
import com.example.holodex.data.db.SyncMetadataEntity
import com.example.holodex.data.db.mappers.toHistoryItemEntity
import com.example.holodex.data.repository.HolodexRepository
import com.example.holodex.viewmodel.mappers.toVideoShell
import java.time.Instant
import javax.inject.Inject

class HistorySynchronizer @Inject constructor(
    private val repository: HolodexRepository,
    private val historyDao: HistoryDao,
    private val syncMetadataDao: SyncMetadataDao,
    private val tokenManager: TokenManager,
    private val logger: SyncLogger
) : ISynchronizer {
    override val name: String = "HISTORY"
    private val METADATA_KEY = "history_last_sync_timestamp"

    override suspend fun synchronize(): Boolean {
        logger.startSection(name)
        val userId = tokenManager.getUserId()
        if (userId.isNullOrBlank()) {
            logger.warning("User ID not found, skipping history sync.")
            logger.endSection(name, success = true) // Success because there's nothing to do
            return true
        }
```

```
try {
    logger.info("Phase 1: Upstream (handled by real-time tracking).")

    logger.info("Phase 2: Fetching remote history playlist...")
    val historyPlaylistId = ":history[user_id=$userId]"
    val remoteResult = repository.getFullPlaylistContent(historyPlaylistId)

    if (remoteResult.isFailure) {
        throw remoteResult.exceptionOrNull() ?: Exception("Failed to fetch remote hist
    }

    val remotePlaylist = remoteResult.getOrThrow()
    val remoteTimestampStr = remotePlaylist.updatedAt
    if (remoteTimestampStr.isNullOrBlank()) {
        throw IllegalStateException("Remote history playlist has no 'updated_at' times
    }

    val remoteTimestamp = Instant.parse(remoteTimestampStr).toEpochMilli()
    val localTimestamp = syncMetadataDao.getLastSyncTimestamp(METADATA_KEY) ?: 0L
    logger.info("  -> Remote Timestamp: $remoteTimestamp | Local Timestamp: $localTime

    if (remoteTimestamp > localTimestamp) {
        logger.info("Phase 3: Server state is newer. Updating local cache.")

        // --- START OF FIX: Generate unique, ordered timestamps ---
        val baseTimestamp = System.currentTimeMillis()

        val remoteHistoryItems = remotePlaylist.content?.mapIndexedNotNull { index, so
            val videoShell = song.toVideoShell(remotePlaylist.title)
            // Pass the synthetic timestamp to the mapper
            song.toHistoryItemEntity(videoShell, baseTimestamp - index)
        } ?: emptyList()
        // --- END OF FIX ---

        historyDao.clearAll()
        // The DAO needs an insertAll method for efficiency
        // If it doesn't have one, this loop is the fallback.
        remoteHistoryItems.forEach { historyDao.insert(it) }

        syncMetadataDao.setLastSyncTimestamp(
            SyncMetadataEntity(
                dataType = METADATA_KEY,
                lastSyncTimestamp = remoteTimestamp
            )
        )
        logger.info("  -> Successfully updated local history with ${remoteHistoryItems
    } else {
        logger.info("Phase 3: Local state is up-to-date or newer. No downstream sync n
    }

    logger.endSection(name, success = true)
    return true

} catch (e: Exception) {
    logger.error(e, "History sync failed catastrophically.")
```

```
            logger.endSection(name, success = false)
            return false
        }
    }
}

// File: java\com\example\holodex\background\ISynchronizer.kt
// File: java/com/example/holodex/background/ISynchronizer.kt (NEW FILE)
package com.example.holodex.background

/**
 * Defines the contract for a class that can synchronize a specific type of data
 * between the local database and a remote server.
 */
interface ISynchronizer {
    /**
     * The unique name of this synchronizer, used for logging.
     */
    val name: String

    /**
     * Executes the full synchronization logic for this data type.
     * @return `true` if the synchronization was successful, `false` otherwise.
     */
    suspend fun synchronize(): Boolean
}

// File: java\com\example\holodex\background\LikesSynchronizer.kt
package com.example.holodex.background

import com.example.holodex.data.db.LikedItemEntity
import com.example.holodex.data.db.SyncStatus
import com.example.holodex.data.repository.HolodexRepository
import javax.inject.Inject

class LikesSynchronizer @Inject constructor(
    private val repository: HolodexRepository,
    private val logger: SyncLogger
) : ISynchronizer {
    override val name: String = "LIKES"

    // Timeout period after which we trust the server's state over a local PENDING_DELETE.
    private val PENDING_DELETE_TIMEOUT_MS = 35 * 60 * 1000L // 35 minutes

    override suspend fun synchronize(): Boolean {
        logger.startSection(name)
        try {
            // --- PHASE 0: PRE-SYNC REPAIR (Remains the same) ---
            logger.info("Phase 0: Checking for orphaned local likes to repair...")
            val orphanedLikes = repository.getOrphanedDirtyLikes() // This DAO method needs to
            if (orphanedLikes.isNotEmpty()) {
                logger.info("  -> Found ${orphanedLikes.size} orphaned items. Attempting to re
                for (orphan in orphanedLikes) {

                    if (orphan.itemType == com.example.holodex.data.db.LikedItemType.SONG_SEGM
```

```
            // --- Logic for Song Segments (Fetch Server ID) ---
            val videoId = orphan.itemId.substringBeforeLast('_')
            val startTime = orphan.itemId.substringAfterLast('_').toIntOrNull()

            if (startTime == null) {
                logger.warning("  -> SKIPPING repair for song segment with malform
                continue
            }

            val result = repository.fetchVideoAndFindSong(videoId, startTime)
            if (result != null && result.second?.id != null) {
                val serverId = result.second!!.id!!
                val repairedItem = orphan.copy(serverId = serverId)
                repository.updateLike(repairedItem)
                logger.logItemAction(LogAction.RECONCILE_SKIP, orphan.actualSongNa
            } else {
                logger.warning("  -> FAILED repair for song '${orphan.actualSongNa
            }
        } else if (orphan.itemType == com.example.holodex.data.db.LikedItemType.VI
            // --- Logic for Videos (Mark as SYNCED) ---
            // Video likes are local-only and should never be DIRTY. This is a dat
            // The correct repair is to mark it as SYNCED.
            val repairedItem = orphan.copy(syncStatus = com.example.holodex.data.d
            repository.updateLike(repairedItem)
            logger.logItemAction(LogAction.RECONCILE_SKIP, orphan.titleSnapshot, o
        }
    }
} else {
    logger.info("  -> No orphaned likes found.")
}
logger.info("Phase 0 complete.")

// --- PHASE 1: UPSTREAM (Client informs the server of its desired state) ---
logger.info("Phase 1: Pushing local changes to server...")
repository.performUpstreamLikesSync(logger)
logger.info("Phase 1 complete.")

// --- PHASE 2: DOWNSTREAM FETCH & STATE RECONCILIATION ---
logger.info("Phase 2: Fetching states and reconciling...")
val remoteLikes = repository.getRemoteLikes()
val allLocalLikes = repository.getAllLocalLikes()
logger.info("  -> Fetched ${remoteLikes.size} remote likes and ${allLocalLikes.siz

val remoteServerIds = remoteLikes.map { it.id }.toSet()
val localServerIdMap = allLocalLikes.filter { it.serverId != null }.associateBy {
val now = System.currentTimeMillis()

val itemsToUpdate = mutableListOf<LikedItemEntity>()
val itemsToDelete = mutableListOf<String>()

// --- RULE 3.1: Item exists on server but not locally at all -> ADD LOCALLY ---
val newFromServer = remoteLikes.filter { !localServerIdMap.containsKey(it.id) }
if (newFromServer.isNotEmpty()) {
    logger.info("  Found ${newFromServer.size} new likes from another device to in
    newFromServer.forEach { p -> logger.logItemAction(LogAction.DOWNSTREAM_INSERT_
```

```
            repository.insertRemoteLikesAsSynced(newFromServer)
        }

        // Iterate through all local items to apply the other rules.
        for (localItem in allLocalLikes) {
            // Rule 3.2 is implicitly handled by the `newFromServer` filter above.
            // We only need to process SYNCED and PENDING_DELETE items here.

            val serverHasItem = localItem.serverId in remoteServerIds

            when (localItem.syncStatus) {
                SyncStatus.SYNCED -> {
                    // --- RULE 3.3: Item is SYNCED locally but NOT on server -> DELETE LO
                    if (!serverHasItem && localItem.serverId != null) {
                        itemsToDelete.add(localItem.itemId)
                        logger.logItemAction(LogAction.DOWNSTREAM_DELETE_LOCAL, localItem.
                    }
                }
                SyncStatus.PENDING_DELETE -> {
                    // --- RULE 3.4: Item is PENDING_DELETE and NOT on server -> DELETE LO
                    if (!serverHasItem) {
                        itemsToDelete.add(localItem.itemId)
                        logger.logItemAction(LogAction.UPSTREAM_DELETE_SUCCESS, localItem.
                    }
                    // --- RULE 4: PENDING_DELETE has timed out AND is still on server ->
                    else if (serverHasItem && now - localItem.lastModifiedAt > PENDING_DEL
                        itemsToUpdate.add(localItem.copy(syncStatus = SyncStatus.SYNCED))
                        logger.logItemAction(LogAction.RECONCILE_SKIP, localItem.actualSon
                    }
                }
                SyncStatus.DIRTY -> {
                    // DIRTY items are handled by the upstream sync and subsequent reconci
                    // We do not need to do anything with them in this loop.
                }
            }
        }

        // Apply all collected changes to the database at once.
        if (itemsToUpdate.isNotEmpty()) {
            logger.info("  -> Updating ${itemsToUpdate.size} items in the database.")
            repository.updateLikesBatch(itemsToUpdate)
        }
        if (itemsToDelete.isNotEmpty()) {
            logger.info("  -> Deleting ${itemsToDelete.size} items from the database.")
            repository.deleteLikesBatch(itemsToDelete)
        }

        logger.info("Phase 2 complete.")
        logger.endSection(name, success = true)
        return true
    } catch (e: Exception) {
        logger.error(e, "Likes sync failed catastrophically.")
        logger.endSection(name, success = false)
        return false
    }
```

```
    }
}

// File: java\com\example\holodex\background\M4AExportWorker.kt
// File: java\com\example\holodex\background\M4AExportWorker.kt
package com.example.holodex.background

import android.content.ContentValues
import android.content.Context
import android.media.MediaMetadataRetriever
import android.net.Uri
import android.os.Build
import android.os.Environment
import android.provider.MediaStore
import androidx.annotation.OptIn
import androidx.core.net.toUri
import androidx.hilt.work.HiltWorker
import androidx.media3.common.MediaItem
import androidx.media3.common.MimeTypes
import androidx.media3.common.util.Clock
import androidx.media3.common.util.UnstableApi
import androidx.media3.datasource.DefaultHttpDataSource
import androidx.media3.datasource.cache.CacheDataSource
import androidx.media3.datasource.cache.SimpleCache
import androidx.media3.exoplayer.source.DefaultMediaSourceFactory
import androidx.media3.transformer.Composition
import androidx.media3.transformer.DefaultDecoderFactory
import androidx.media3.transformer.ExoPlayerAssetLoader
import androidx.media3.transformer.ExportException
import androidx.media3.transformer.ExportResult
import androidx.media3.transformer.InAppMp4Muxer
import androidx.media3.transformer.Transformer
import androidx.work.CoroutineWorker
import androidx.work.WorkerParameters
import com.example.holodex.data.db.DownloadStatus
import com.example.holodex.data.db.DownloadedItemDao
import com.example.holodex.di.DownloadCache
import dagger.assisted.Assisted
import dagger.assisted.AssistedInject
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.suspendCancellableCoroutine
import kotlinx.coroutines.withContext
import timber.log.Timber
import java.io.File
import java.io.IOException
import java.net.URLDecoder
import kotlin.coroutines.resume
import kotlin.coroutines.resumeWithException

@OptIn(UnstableApi::class)
@HiltWorker
class M4AExportWorker @AssistedInject constructor(
    @Assisted private val context: Context,
    @Assisted workerParams: WorkerParameters,
    @DownloadCache private val downloadCache: SimpleCache,
```

```
    private val downloadedItemDao: DownloadedItemDao,
    private val metadataWriter: MetadataWriter // <-- INJECT the MetadataWriter
) : CoroutineWorker(context, workerParams) {

    companion object {
        // Keys for WorkManager Data
        const val KEY_ITEM_ID = "ITEM_ID"
        const val KEY_ORIGINAL_URI = "ORIGINAL_URI"
        const val KEY_SONG_TITLE = "SONG_TITLE"
        const val KEY_ARTIST_NAME = "ARTIST_NAME"
        const val KEY_ALBUM_NAME = "ALBUM_NAME"
        const val KEY_ARTWORK_URI = "ARTWORK_URI"
        const val KEY_CLIP_START_MS = "CLIP_START_MS"
        const val KEY_CLIP_END_MS = "CLIP_END_MS"
        const val KEY_TRACK_NUMBER = "TRACK_NUMBER"
        private const val TAG = "M4AExportWorker"
    }

    override suspend fun doWork(): Result {
        val itemId = inputData.getString(KEY_ITEM_ID) ?: return Result.failure()
        val originalUriString = inputData.getString(KEY_ORIGINAL_URI) ?: return Result.failure
        val songTitle = inputData.getString(KEY_SONG_TITLE) ?: "Unknown Title"
        val artistName = inputData.getString(KEY_ARTIST_NAME) ?: "Unknown Artist"
        val albumName = inputData.getString(KEY_ALBUM_NAME) ?: "Unknown Album"
        val artworkUri = inputData.getString(KEY_ARTWORK_URI)
        val clipStartMs = inputData.getLong(KEY_CLIP_START_MS, -1)
        val clipEndMs = inputData.getLong(KEY_CLIP_END_MS, -1)
        val trackNumber = inputData.getInt(KEY_TRACK_NUMBER, -1)

        if (clipStartMs == -1L || clipEndMs == -1L) {
            Timber.e("$TAG: Invalid clip times provided.")
            downloadedItemDao.updateStatus(itemId, DownloadStatus.FAILED)
            return Result.failure()
        }

        val decodedTitle = try {
            URLDecoder.decode(songTitle, "UTF-8")
        } catch (_: Exception) {
            songTitle
        }
        val sanitizedDisplayTitle = decodedTitle.replace(Regex("[\\\\/:*?\"<>|]"), "_").take(1
        val finalFileName = "$sanitizedDisplayTitle.m4a"

        val tempOutputFile = File(context.cacheDir, "transformer_output_${itemId}.m4a")

        Timber.d("$TAG: Starting export for item: $itemId, filename: $finalFileName")

        try {
            // Step 1: Use Transformer to create a clean, clipped M4A file
            val mediaItem = MediaItem.Builder()
                .setUri(originalUriString.toUri())
                .setClippingConfiguration(
                    MediaItem.ClippingConfiguration.Builder()
                        .setStartPositionMs(clipStartMs)
                        .setEndPositionMs(clipEndMs)
```

```
                .build()
        )
        .build()

    val exportResult = withContext(Dispatchers.Main) {
        val transformer = buildTransformerOnMainThread()
        transformMediaItem(transformer, mediaItem, tempOutputFile.absolutePath)
    }

    if (exportResult.exportException != null) throw exportResult.exportException!!
    Timber.d("$TAG: Transformer successfully created temp file: ${tempOutputFile.absol

    // Step 2: Write metadata to the temporary file using the injected MetadataWriter
    metadataWriter.writeMetadata(
        context = context,
        targetFile = tempOutputFile,
        itemId = itemId,
        songTitle = songTitle,
        artistName = artistName,
        albumTitle = albumName,
        artworkUrl = artworkUri,
        trackNumber = trackNumber
    )
    Timber.d("$TAG: MetadataWriter successfully wrote tags to temp file.")


    // Step 3: Verify and export the now-tagged file
    verifyMetadataInFile(
        tempOutputFile,
        songTitle,
        artistName,
        albumName,
        artworkUri != null
    )
    val finalUri = exportToMediaStore(tempOutputFile, finalFileName)
        ?: throw IOException("Failed to export temp file to MediaStore.")

    // Step 4: Finalize and clean up
    downloadedItemDao.updateStatusToCompleted(
        videoId = itemId,
        uri = finalUri.toString(),
        fileName = finalFileName, // <-- FIX: Pass the file name
        timestamp = System.currentTimeMillis() // <-- FIX: Pass the timestamp
    )
    downloadCache.removeResource(itemId)
    tempOutputFile.delete()

    Timber.i("$TAG: Successfully exported item $itemId to $finalUri")
    return Result.success()

} catch (e: Exception) {
    Timber.e(e, "$TAG: Export failed for item $itemId.")
    downloadedItemDao.updateStatus(itemId, DownloadStatus.FAILED)
    tempOutputFile.delete()
    return Result.failure()
```

```
    }
}

private fun buildTransformerOnMainThread(): Transformer {
    val upstreamDataSourceFactory = DefaultHttpDataSource.Factory()
    val cacheDataSourceFactory = CacheDataSource.Factory()
        .setCache(downloadCache)
        .setUpstreamDataSourceFactory(upstreamDataSourceFactory)
    val mediaSourceFactory =
        DefaultMediaSourceFactory(context).setDataSourceFactory(cacheDataSourceFactory)
    val assetLoaderFactory = ExoPlayerAssetLoader.Factory(
        context,
        DefaultDecoderFactory.Builder(context).build(),
        Clock.DEFAULT,
        mediaSourceFactory
    )

    return Transformer.Builder(context)
        .setAssetLoaderFactory(assetLoaderFactory)
        .setAudioMimeType(MimeTypes.AUDIO_AAC)
        .experimentalSetMp4EditListTrimEnabled(true)
        .setMuxerFactory(InAppMp4Muxer.Factory())
        .build()
}

private suspend fun transformMediaItem(
    transformer: Transformer,
    mediaItem: MediaItem,
    outputPath: String
): ExportResult {
    return suspendCancellableCoroutine { continuation ->
        val listener = object : Transformer.Listener {
            override fun onCompleted(composition: Composition, exportResult: ExportResult)
                if (continuation.isActive) continuation.resume(exportResult)
            }

            override fun onError(
                composition: Composition,
                exportResult: ExportResult,
                exportException: ExportException
            ) {
                if (continuation.isActive) continuation.resumeWithException(exportExceptio
            }
        }
        transformer.addListener(listener)
        continuation.invokeOnCancellation {
            transformer.removeListener(listener)
            transformer.cancel()
        }
        transformer.start(mediaItem, outputPath)
    }
}
```

```
private fun exportToMediaStore(sourceFile: File, finalFileName: String): Uri? {
    val resolver = context.contentResolver
    val contentValues = ContentValues().apply {
        put(MediaStore.MediaColumns.DISPLAY_NAME, finalFileName)
        put(MediaStore.MediaColumns.MIME_TYPE, "audio/mp4")
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q) {
            put(
                MediaStore.MediaColumns.RELATIVE_PATH,
                Environment.DIRECTORY_MUSIC + File.separator + "HolodexMusic"
            )
            put(MediaStore.Audio.Media.IS_PENDING, 1)
        } else {
            @Suppress("DEPRECATION")
            val musicDir =
                Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_MUSIC)
            val appMusicDir = File(musicDir, "HolodexMusic")
            if (!appMusicDir.exists() && !appMusicDir.mkdirs()) {
                return null
            }
            val targetFile = File(appMusicDir, finalFileName)
            put(MediaStore.MediaColumns.DATA, targetFile.absolutePath)
        }
    }
    val collection = if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q) {
        MediaStore.Audio.Media.getContentUri(MediaStore.VOLUME_EXTERNAL_PRIMARY)
    } else {
        @Suppress("DEPRECATION")
        MediaStore.Audio.Media.EXTERNAL_CONTENT_URI
    }
    val uri = resolver.insert(collection, contentValues)
    uri?.let { outputUri ->
        try {
            resolver.openOutputStream(outputUri)?.use { outputStream ->
                sourceFile.inputStream().use { inputStream ->
                    inputStream.copyTo(outputStream)
                }
            }
            if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q) {
                contentValues.clear()
                contentValues.put(MediaStore.Audio.Media.IS_PENDING, 0)
                resolver.update(outputUri, contentValues, null, null)
            }
            return outputUri
        } catch (e: Exception) {
            Timber.e(e, "Failed to copy file to MediaStore for $finalFileName.")
            resolver.delete(outputUri, null, null)
        }
    }
    return null
}

private fun verifyMetadataInFile(
    file: File,
    expectedTitle: String,
    expectedArtist: String,
```

```
        expectedAlbum: String,
        shouldHaveArtwork: Boolean
    ) {
        var retriever: MediaMetadataRetriever? = null
        try {
            retriever = MediaMetadataRetriever()
            retriever.setDataSource(file.absolutePath)
            val foundTitle = retriever.extractMetadata(MediaMetadataRetriever.METADATA_KEY_TIT
            val foundArtist = retriever.extractMetadata(MediaMetadataRetriever.METADATA_KEY_AR
            val foundAlbum = retriever.extractMetadata(MediaMetadataRetriever.METADATA_KEY_ALB
            val hasArtwork = retriever.embeddedPicture?.isNotEmpty() == true
            val titleMatches = foundTitle == expectedTitle
            val artistMatches = foundArtist == expectedArtist
            val albumMatches = foundAlbum == expectedAlbum
            val artworkMatches = if (shouldHaveArtwork) hasArtwork else true
            if (titleMatches && artistMatches && albumMatches && artworkMatches) {
                Timber.i("Metadata Verification PASSED - All expected metadata found correctly
            } else {
                Timber.w("--- Metadata Verification FAILED ---")
                if (!titleMatches) Timber.w("--> Title Mismatch: Expected '$expectedTitle', Fo
                if (!artistMatches) Timber.w("--> Artist Mismatch: Expected '$expectedArtist',
                if (!albumMatches) Timber.w("--> Album Mismatch: Expected '$expectedAlbum', Fo
                if (!artworkMatches) Timber.w("--> Artwork Mismatch: Expected artwork to be pr
                Timber.w("-----------------------------------")
            }
        } catch (e: Exception) {
            Timber.e(e, "Metadata verification failed due to an exception.")
        } finally {
            try {
                retriever?.release()
            } catch (e: Exception) {
                Timber.w(e, "Failed to release MediaMetadataRetriever")
            }
        }
    }
}

// File: java\com\example\holodex\background\MetadataUpdateWorker.kt
// File: java/com/example/holodex/background/MetadataUpdateWorker.kt

package com.example.holodex.background

import android.content.Context
import android.graphics.Bitmap
import androidx.core.net.toUri
import androidx.hilt.work.HiltWorker
import androidx.work.CoroutineWorker
import androidx.work.WorkerParameters
import coil.imageLoader
import coil.request.ImageRequest
import coil.size.Size
import com.example.holodex.data.db.DownloadedItemDao
import dagger.assisted.Assisted
import dagger.assisted.AssistedInject
import kotlinx.coroutines.Dispatchers
```

```
import kotlinx.coroutines.withContext
import org.jaudiotagger.audio.AudioFileIO
import org.jaudiotagger.tag.FieldKey
import org.jaudiotagger.tag.images.ArtworkFactory
import timber.log.Timber
import java.io.ByteArrayOutputStream
import java.io.File
import java.io.IOException

@HiltWorker
class MetadataUpdateWorker @AssistedInject constructor(
    @Assisted private val context: Context,
    @Assisted workerParams: WorkerParameters,
    private val downloadedItemDao: DownloadedItemDao
) : CoroutineWorker(context, workerParams) {

    companion object {
        // Keys for WorkManager Data
        const val KEY_ITEM_ID = "ITEM_ID"
        const val KEY_FILE_URI = "FILE_URI"
        const val KEY_SONG_TITLE = "SONG_TITLE"
        const val KEY_ARTIST_NAME = "ARTIST_NAME"
        const val KEY_ALBUM_NAME = "ALBUM_NAME"
        const val KEY_ARTWORK_URI = "ARTWORK_URI"
        const val KEY_TRACK_NUMBER = "TRACK_NUMBER"
        private const val TAG = "MetadataUpdateWorker"
    }

    override suspend fun doWork(): Result = withContext(Dispatchers.IO) {
        val itemId = inputData.getString(KEY_ITEM_ID) ?: return@withContext Result.failure()
        val fileUriString = inputData.getString(KEY_FILE_URI) ?: return@withContext Result.fai
        val songTitle = inputData.getString(KEY_SONG_TITLE)
        val artistName = inputData.getString(KEY_ARTIST_NAME)
        val albumName = inputData.getString(KEY_ALBUM_NAME)
        val artworkUri = inputData.getString(KEY_ARTWORK_URI)
        val trackNumber = inputData.getInt(KEY_TRACK_NUMBER, -1)

        val tempFile = File(context.cacheDir, "metadata_update_${itemId}.m4a")

        try {
            val originalUri = fileUriString.toUri()
            Timber.d("$TAG: Starting metadata update for $itemId at URI: $originalUri")

            // 1. Copy original file to a temporary location for safe editing.
            context.contentResolver.openInputStream(originalUri)?.use { input ->
                tempFile.outputStream().use { output ->
                    input.copyTo(output)
                }
            } ?: throw IOException("Could not open input stream for original file.")

            // 2. Fetch new artwork if available.
            val artworkBytes = fetchArtwork(artworkUri)

            // 3. Use JAudioTagger to write new metadata to the temporary file.
            val audioFile = AudioFileIO.read(tempFile)
```

```
            val tag = audioFile.tagOrCreateAndSetDefault
            if (songTitle != null) tag.setField(FieldKey.TITLE, songTitle)
            if (artistName != null) tag.setField(FieldKey.ARTIST, artistName)
            if (albumName != null) tag.setField(FieldKey.ALBUM, albumName)
            if (trackNumber > 0) tag.setField(FieldKey.TRACK, trackNumber.toString())
            if (artworkBytes != null) {
                tag.deleteArtworkField()
                val artwork = ArtworkFactory.getNew()
                artwork.setBinaryData(artworkBytes)
                artwork.mimeType = "image/jpeg"
                tag.setField(artwork)
            }
            AudioFileIO.write(audioFile)
            Timber.d("$TAG: Successfully wrote new tags to temp file.")

            // 4. Overwrite the original file with the newly tagged temporary file.
            context.contentResolver.openOutputStream(originalUri, "w")?.use { output ->
                tempFile.inputStream().use { input ->
                    input.copyTo(output)
                }
            } ?: throw IOException("Could not open output stream to overwrite original file.")

            Timber.i("$TAG: Successfully updated metadata for $itemId.")
            return@withContext Result.success()

        } catch (e: Exception) {
            Timber.e(e, "$TAG: Failed to update metadata for $itemId.")
            return@withContext Result.failure()
        } finally {
            tempFile.delete()
        }
    }

    private suspend fun fetchArtwork(url: String?): ByteArray? {
        if (url == null) return null
        val highResUrl = url.replace(Regex("""/100x100bb\.jpg$"""), "/1000x1000bb.jpg")
        return try {
            val request = ImageRequest.Builder(context).data(highResUrl).size(Size(1000, 1000)
                .allowHardware(false).build()
            (context.imageLoader.execute(request).drawable as? android.graphics.drawable.Bitma
                val stream = ByteArrayOutputStream()
                bitmap.compress(Bitmap.CompressFormat.JPEG, 95, stream)
                stream.toByteArray()
            }
        } catch (e: Exception) {
            Timber.e(e, "Failed to fetch artwork from $highResUrl for metadata update.")
            null
        }
    }
}

// File: java\com\example\holodex\background\MetadataWriter.kt
package com.example.holodex.background

import android.content.Context
```

```
import android.graphics.Bitmap
import android.graphics.drawable.BitmapDrawable
import coil.imageLoader
import coil.request.ImageRequest
import coil.size.Size
import org.jaudiotagger.audio.AudioFileIO
import org.jaudiotagger.tag.FieldKey
import org.jaudiotagger.tag.TagOptionSingleton
import org.jaudiotagger.tag.images.ArtworkFactory
import timber.log.Timber
import java.io.ByteArrayOutputStream
import java.io.File
import java.io.IOException
import javax.inject.Inject
import javax.inject.Singleton

@Singleton
class MetadataWriter @Inject constructor() {

    private val tag = "MetadataWriter"

    init {
        // Global configuration for JAudioTagger
        TagOptionSingleton.getInstance().isPadNumbers = false
    }

    suspend fun writeMetadata(
        context: Context,
        targetFile: File,
        itemId: String,
        songTitle: String,
        artistName: String,
        albumTitle: String,
        artworkUrl: String?,
        trackNumber: Int
    ) {
        try {
            Timber.d("$tag: Starting metadata tagging for ${targetFile.name}")
            val artworkData = fetchArtwork(context, artworkUrl)

            val audioFile = AudioFileIO.read(targetFile)
            val tag = audioFile.tagOrCreateAndSetDefault

            tag.setField(FieldKey.TITLE, songTitle)
            tag.setField(FieldKey.COMMENT, "holodex_item_id::$itemId")
            tag.setField(FieldKey.ARTIST, artistName)
            tag.setField(FieldKey.ALBUM, albumTitle)
            tag.setField(FieldKey.ALBUM_ARTIST, artistName)
            if (trackNumber > 0) {
                tag.setField(FieldKey.TRACK, trackNumber.toString())
            }

            if (artworkData != null) {
                tag.deleteArtworkField()
                val artwork = ArtworkFactory.getNew()
```

```
                artwork.setBinaryData(artworkData)
                artwork.mimeType = "image/jpeg"
                artwork.description = "Cover"
                tag.setField(artwork)
            }
            AudioFileIO.write(audioFile)
            Timber.i("$tag: Successfully wrote tags to ${targetFile.name}")

        } catch (e: Exception) {
            throw IOException("JAudioTagger tagging failed.", e)
        }
    }

    private suspend fun fetchArtwork(context: Context, url: String?): ByteArray? {
        if (url == null) return null
        val highResUrl = url.replace(Regex("""/100x100bb\.jpg$"""), "/1000x1000bb.jpg")
        return try {
            val request = ImageRequest.Builder(context)
                .data(highResUrl)
                .size(Size(1000, 1000))
                .allowHardware(false)
                .build()
            (context.imageLoader.execute(request).drawable as? BitmapDrawable)?.bitmap?.let {
                val stream = ByteArrayOutputStream()
                bitmap.compress(Bitmap.CompressFormat.JPEG, 95, stream)
                stream.toByteArray()
            }
        } catch (e: Exception) {
            Timber.e(e, "$tag: Failed to fetch artwork from $highResUrl")
            null
        }
    }
}

// File: java\com\example\holodex\background\PlaylistSynchronizer.kt
// File: java/com/example/holodex/background/PlaylistSynchronizer.kt
package com.example.holodex.background

import com.example.holodex.data.db.SyncStatus
import com.example.holodex.data.repository.HolodexRepository
import timber.log.Timber
import javax.inject.Inject

class PlaylistSynchronizer @Inject constructor(
    private val repository: HolodexRepository,
    private val logger: SyncLogger
) : ISynchronizer {
    override val name: String = "PLAYLISTS"

    override suspend fun synchronize(): Boolean {
        logger.startSection(name)
        try {
            // --- PHASE 1: UPSTREAM (Client -> Server) ---
            logger.info("Phase 1: Pushing local changes to server...")
            repository.performUpstreamPlaylistDeletions(logger)
```

```
repository.performUpstreamPlaylistUpserts(logger)
logger.info("Phase 1 complete.")

// --- PHASE 2: FETCH ---
logger.info("Phase 2: Fetching remote and local states...")
val remotePlaylists = repository.getRemotePlaylists()
val localPlaylists = repository.getLocalPlaylists()
logger.info("  -> Fetched ${remotePlaylists.size} remote playlists and ${localPlay

// --- PHASE 3: METADATA RECONCILIATION ---
logger.info("Phase 3: Reconciling playlist metadata...")
val remoteMap = remotePlaylists.associateBy { it.serverId }
val localMap = localPlaylists.filter { it.serverId != null }.associateBy { it.serv

val newFromServer = remotePlaylists.filter { !localMap.containsKey(it.serverId) }
if (newFromServer.isNotEmpty()) {
    logger.info("  Found ${newFromServer.size} new playlists from server:")
    newFromServer.forEach { p -> logger.logItemAction(LogAction.DOWNSTREAM_INSERT_
    repository.insertNewSyncedPlaylists(newFromServer)
}

val deletedOnServer = localPlaylists.filter {
    it.serverId != null && it.syncStatus == SyncStatus.SYNCED && !remoteMap.contai
}
if (deletedOnServer.isNotEmpty()) {
    logger.info("  Found ${deletedOnServer.size} playlists deleted on server:")
    deletedOnServer.forEach { p -> logger.logItemAction(LogAction.DOWNSTREAM_DELET
    repository.deleteLocalPlaylists(deletedOnServer.map { it.playlistId })
}
logger.info("Phase 3 complete.")


// --- PHASE 4: CONTENT SYNCHRONIZATION (Timestamp-based) ---
logger.info("Phase 4: Reconciling song content for all user-owned playlists...")
val finalLocalPlaylists = repository.getLocalPlaylists().filter { it.serverId != n

for (localPlaylist in finalLocalPlaylists) {
    val remotePlaylist = remoteMap[localPlaylist.serverId] ?: continue
    if (localPlaylist.syncStatus == SyncStatus.DIRTY) {
        logger.info("  -> Skipping content sync for DIRTY playlist '${localPlaylis
        continue
    }

    val localTimestamp = localPlaylist.last_modified_at
    val remoteTimestamp = remotePlaylist.last_modified_at

    if (localTimestamp != null && remoteTimestamp != null && remoteTimestamp > loc
        logger.info("  -> Server is newer for playlist '${localPlaylist.name}'. Re

        // CRITICAL FIX: Capture count BEFORE any operations
        val itemCountBefore = repository.getPlaylistItemCount(localPlaylist.playli
        val localOnlyCountBefore = repository.getLocalOnlyItemCount(localPlaylist.

        Timber.tag("SYNC_DEBUG").i(
            "BEFORE sync: Playlist '${localPlaylist.name}' has $itemCountBefore to
```

```
                    )

                    // 1. Get the remote content first.
                    val remoteContent = repository.getRemotePlaylistContent(remotePlaylist.ser

                    // 2. Perform the SAFE reconciliation of items FIRST. This preserves local
                    //    This MUST happen before any metadata updates to avoid triggering cas
                    repository.reconcileLocalPlaylistItems(localPlaylist.playlistId, remoteCon

                    // 3. ONLY AFTER the items are safely reconciled, update the parent playli
                    //    This updates the timestamp to match the server, preventing re-sync o
                    repository.updateLocalPlaylistMetadata(localPlaylist.playlistId, remotePla

                    logger.info("    -> Reconciled local content with ${remoteContent.size} se

                } else {
                    logger.info("  -> Local state for playlist '${localPlaylist.name}' is curr
                }
            }
            logger.info("Phase 4 complete.")

            logger.endSection(name, success = true)
            return true
        } catch (e: Exception) {
            logger.error(e, "Playlist sync failed catastrophically.")
            logger.endSection(name, success = false)
            return false
        }
    }
}

// File: java\com\example\holodex\background\StarredPlaylistSynchronizer.kt
// File: java/com/example/holodex/background/StarredPlaylistSynchronizer.kt (NEW FILE)
package com.example.holodex.background

import com.example.holodex.data.repository.HolodexRepository
import javax.inject.Inject

class StarredPlaylistSynchronizer @Inject constructor(
    private val repository: HolodexRepository,
    private val logger: SyncLogger
) : ISynchronizer {
    override val name: String = "STARRED_PLAYLISTS"

    override suspend fun synchronize(): Boolean {
        logger.startSection(name)
        try {
            // --- PHASE 1: UPSTREAM ---
            logger.info("Phase 1: Pushing local changes to server...")
            repository.performUpstreamStarredPlaylistsSync(logger)
            logger.info("Phase 1 complete.")

            // --- PHASE 2: SMART SERVER SYNC ---
            logger.info("Phase 2: Fetching server state and reconciling local data...")
            val remoteStarred = repository.getRemoteStarredPlaylists()
```

```
            logger.info("  -> Fetched ${remoteStarred.size} starred playlists from server.")

            val unsyncedCount = repository.getLocalUnsyncedStarredPlaylistsCount()
            if (unsyncedCount > 0) {
                logger.info("  -> Preserving $unsyncedCount locally-changed items.")
            }

            val deletedCount = repository.deleteLocalSyncedStarredPlaylists()
            logger.info("  -> Wiped $deletedCount SYNCED items from local database.")

            repository.insertRemoteStarredPlaylistsAsSynced(remoteStarred)
            logger.info("  -> Paved local database with ${remoteStarred.size} fresh items from
            logger.info("Phase 2 complete.")

            logger.endSection(name, success = true)
            return true
        } catch (e: Exception) {
            logger.error(e, "Starred Playlists sync failed catastrophically.")
            logger.endSection(name, success = false)
            return false
        }
    }
}

// File: java\com\example\holodex\background\SyncCoordinator.kt
// File: java/com/example/holodex/background/SyncCoordinator.kt (NEW FILE)
package com.example.holodex.background

import kotlinx.coroutines.async
import kotlinx.coroutines.coroutineScope
import javax.inject.Inject
import javax.inject.Singleton

@Singleton
class SyncCoordinator @Inject constructor(
    private val synchronizers: Set<@JvmSuppressWildcards ISynchronizer>,
    private val logger: SyncLogger
) {
    suspend fun run(): Boolean {
        logger.info("===== Starting Full Synchronization Run =====")
        var allSucceeded = true

        try {
            coroutineScope {
                val results = synchronizers.map { synchronizer ->
                    async {
                        // We will run some syncs sequentially if they are dependent in the fu
                        // For now, all are independent.
                        synchronizer.synchronize()
                    }
                }.map { it.await() }

                if (results.contains(false)) {
                    allSucceeded = false
                }
```

```
            }
        } catch (e: Exception) {
            logger.error(e, "The SyncCoordinator caught an unhandled exception.")
            allSucceeded = false
        }

        logger.info("===== Full Synchronization Run Finished. Overall Success: $allSucceeded =
        return allSucceeded
    }
}

// File: java\com\example\holodex\background\SyncLogger.kt
// File: java/com/example/holodex/background/SyncLogger.kt
package com.example.holodex.background

import timber.log.Timber
import javax.inject.Inject
import javax.inject.Singleton

enum class LogAction {
    // Upstream
    UPSTREAM_DELETE_SUCCESS,
    UPSTREAM_DELETE_FAILED,
    UPSTREAM_UPSERT_SUCCESS,
    UPSTREAM_UPSERT_FAILED,
    // Downstream
    DOWNSTREAM_INSERT_LOCAL,
    DOWNSTREAM_DELETE_LOCAL,
    DOWNSTREAM_UPDATE_LOCAL,
    // Reconciliation
    RECONCILE_SKIP
}

@Singleton
class SyncLogger @Inject constructor() {
    private val TAG = "SYNC"

    fun startSection(name: String) {
        Timber.tag(TAG).i("====== STARTING $name SYNC ======")
    }

    fun endSection(name: String, success: Boolean) {
        val status = if (success) "SUCCESSFUL" else "FAILED"
        Timber.tag(TAG).i("====== $name SYNC $status ======")
    }

    fun info(message: String) {
        Timber.tag(TAG).i(message)
    }

    fun warning(message: String) {
        Timber.tag(TAG).w(message)
    }

    fun error(throwable: Throwable, message: String) {
```

```
        Timber.tag(TAG).e(throwable, message)
    }

    fun logItemAction(
        action: LogAction,
        itemName: String?,
        localId: Long?,
        serverId: String?,
        reason: String? = null
    ) {
        val formattedName = "'${itemName ?: "Unknown"}'"
        val formattedIds = "(LID: ${localId ?: "N/A"}, SID: ${serverId ?: "N/A"})"
        val formattedReason = reason?.let { " | Reason: $it" } ?: ""
        Timber.tag(TAG).d("-> [${action.name}] Item: $formattedName $formattedIds$formattedRea
    }
}

// File: java\com\example\holodex\background\SyncWorker.kt
// File: java/com/example/holodex/background/SyncWorker.kt (MODIFIED)
package com.example.holodex.background

import android.content.Context
import androidx.hilt.work.HiltWorker
import androidx.work.CoroutineWorker
import androidx.work.WorkerParameters
import com.example.holodex.auth.TokenManager
import dagger.assisted.Assisted
import dagger.assisted.AssistedInject
import timber.log.Timber

@HiltWorker
class SyncWorker @AssistedInject constructor(
    @Assisted context: Context,
    @Assisted params: WorkerParameters,
    // --- MODIFICATION: Inject SyncCoordinator, not HolodexRepository ---
    private val syncCoordinator: SyncCoordinator,
    private val tokenManager: TokenManager
) : CoroutineWorker(context, params) {

    companion object {
        const val TAG = "SyncWorker"
        const val MAX_RUN_ATTEMPTS = 3
    }

    override suspend fun doWork(): Result {
        Timber.i("$TAG: Starting synchronization work using SyncCoordinator.")

        if (tokenManager.getJwt() == null) {
            Timber.i("$TAG: User not logged in. Sync work is not required. Finishing successfu
            return Result.success()
        }

        return try {
            // --- MODIFICATION: Call the coordinator ---
            val success = syncCoordinator.run()
```

```
            if (success) {
                Timber.i("$TAG: Synchronization work completed successfully.")
                Result.success()
            } else {
                // The coordinator's log will have the details. We just handle the retry logic
                Timber.w("$TAG: SyncCoordinator reported failure on attempt $runAttemptCount."
                if (runAttemptCount < MAX_RUN_ATTEMPTS) {
                    Timber.w("$TAG: Scheduling a retry.")
                    Result.retry()
                } else {
                    Timber.e("$TAG: Maximum retry attempts reached. Failing the work.")
                    Result.failure()
                }
            }
        } catch (e: Exception) {
            Timber.e(e, "$TAG: SyncWorker caught an unhandled exception.")
            if (runAttemptCount < MAX_RUN_ATTEMPTS) Result.retry() else Result.failure()
        }
    }
}

// File: java\com\example\holodex\data\api\AuthenticatedMusicdexApiService.kt
// File: java/com/example/holodex/data/api/AuthenticatedMusicdexApiService.kt

package com.example.holodex.data.api

import com.example.holodex.data.model.discovery.DiscoveryResponse
import com.example.holodex.data.model.discovery.FullPlaylist
import com.example.holodex.data.model.discovery.PlaylistStub
import retrofit2.Response
import retrofit2.http.Body
import retrofit2.http.DELETE
import retrofit2.http.GET
import retrofit2.http.HTTP
import retrofit2.http.PATCH
import retrofit2.http.POST
import retrofit2.http.Path
import retrofit2.http.Query

data class LikeRequest(val song_id: String)
// The incorrect DeleteLikeRequest class is removed.

data class StarPlaylistRequest(val playlist_id: String)

data class LikedSongApiDto(
    val id: String, // The song's unique ID
    val channel_id: String,
    val video_id: String,
    val name: String,
    val start: Int,
    val end: Int,
    val original_artist: String?,
    val art: String?,
    val channel: ApiChannelStub?
```

```
)

data class ApiChannelStub(
    val name: String,
    val english_name: String?,
    val photo: String?
)

data class PaginatedLikesResponse(
    val page_count: Int,
    val content: List<LikedSongApiDto>
)

data class FavoriteChannelApiDto(
    val id: String,
    val name: String? = null,
    val english_name: String? = null,
    val photo: String?,
    val org: String?,
    val twitter: String?
)
data class PatchOperation(
    val op: String, // "add" or "remove"
    val channel_id: String
)
typealias PatchFavoriteChannelsRequest = List<PatchOperation>

/**
 * Defines the authenticated endpoints for the music.holodex.net API.
 */
interface AuthenticatedMusicdexApiService {

    @GET("api/v2/musicdex/discovery/favorites")
    suspend fun getDiscoveryForFavorites(): Response<DiscoveryResponse>

    @GET("api/v2/musicdex/like")
    suspend fun getLikes(
        @Query("since") sinceTimestamp: Long? = null,
        @Query("page") page: Int? = 1,
        @Query("paginated") paginated: Boolean = true
    ): Response<PaginatedLikesResponse>

    @GET("api/v2/musicdex/like/check")
    suspend fun checkLikes(@Query("song_id") songIds: String): Response<List<Boolean>>

    @POST("api/v2/musicdex/like")
    suspend fun addLike(@Body request: LikeRequest): Response<Unit>

    // --- FIX: Change to a proper DELETE request with a body ---
    @HTTP(method = "DELETE", path = "api/v2/musicdex/like", hasBody = true)
    suspend fun deleteLike(@Body request: LikeRequest): Response<Unit>
    // --- END OF FIX ---

    @PATCH("api/v2/users/favorites")
    suspend fun patchFavoriteChannels(@Body request: PatchFavoriteChannelsRequest): Response<L
```

```
    @GET("api/v2/musicdex/history/{songId}")
    suspend fun trackSongInHistory(@Path("songId") songId: String): Response<Unit>

    @GET("api/v2/musicdex/playlist/{playlistId}")
    suspend fun getPlaylistContent(@Path(value = "playlistId", encoded = true) playlistId: Str

    @GET("api/v2/musicdex/playlist")
    suspend fun getMyPlaylists(): Response<List<PlaylistDto>>

    @POST("api/v2/musicdex/playlist")
    suspend fun createOrUpdatePlaylist(@Body playlist: PlaylistUpdateRequest): Response<List<P

    @DELETE("api/v2/musicdex/playlist/{playlistId}")
    suspend fun deletePlaylist(@Path("playlistId") playlistId: String): Response<Unit>

    @GET("api/v2/musicdex/playlist/{playlistId}/{songId}")
    suspend fun addSongToPlaylist(
        @Path("playlistId") playlistId: String,
        @Path("songId") songId: String
    ): Response<Unit>

    @DELETE("api/v2/musicdex/playlist/{playlistId}/{songId}")
    suspend fun removeSongFromPlaylist(
        @Path("playlistId") playlistId: String,
        @Path("songId") songId: String
    ): Response<Unit>

    @GET("api/v2/musicdex/star")
    suspend fun getStarredPlaylists(): Response<List<PlaylistStub>>

    @POST("api/v2/musicdex/star")
    suspend fun starPlaylist(@Body request: StarPlaylistRequest): Response<Unit>

    @HTTP(method = "DELETE", path = "api/v2/musicdex/star", hasBody = true)
    suspend fun unstarPlaylist(@Body request: StarPlaylistRequest): Response<Unit>
}

// File: java\com\example\holodex\data\api\HolodexApiService.kt
// File: java\com\example\holodex\data\api\HolodexApiService.kt

package com.example.holodex.data.api

import com.example.holodex.data.model.HolodexVideoItem
import com.example.holodex.data.model.PaginatedVideosResponse
import com.example.holodex.data.model.VideoSearchRequest
import com.example.holodex.data.model.discovery.ChannelDetails
import com.example.holodex.data.model.discovery.MusicdexSong
import com.google.gson.annotations.SerializedName
import retrofit2.Response
import retrofit2.http.Body
import retrofit2.http.GET
import retrofit2.http.POST
import retrofit2.http.Path
import retrofit2.http.Query
```

```
// --- Request/Response Models ---
data class LoginRequest(val service: String, val token: String)
data class LoginResponse(val jwt: String) // Assuming user object is handled separately
data class LatestSongsRequest(
    val channel_id: String? = null,
    val paginated: Boolean = true,
    val limit: Int = 25,
    val offset: Int = 0
)

data class PaginatedSongsResponse(
    val total: String,
    val items: List<MusicdexSong>
) {
    fun getTotalAsInt(): Int? = total.toIntOrNull()
}

// Represents the structure from /statics/orgs.json
data class Organization(
    val name: String,
    @SerializedName("name_jp") val nameJp: String?,
    val short: String?
)

data class PaginatedChannelsResponse(
    @SerializedName("total") val total: String?,
    @SerializedName("items") val items: List<ChannelDetails>
) {
    fun getTotalAsInt(): Int? = total?.toIntOrNull()
}

interface HolodexApiService {

    @GET("api/v2/videos/{videoId}")
    suspend fun getVideoWithSongs(
        @Path("videoId") videoId: String,
        @Query("include") include: String = "songs,live_info,description",
        @Query("lang") lang: String = "en",
        @Query("c") comments: String? = null
    ): Response<HolodexVideoItem>

    @POST("api/v2/search/videoSearch")
    suspend fun searchVideosAdvanced(
        @Body request: VideoSearchRequest
    ): Response<PaginatedVideosResponse>

    @POST("api/v2/user/login")
    suspend fun login(@Body request: LoginRequest): Response<LoginResponse>

    @GET("api/v2/user/refresh")
    suspend fun refreshUser(): Response<LoginResponse> // Assuming it returns a new JWT

    @GET("api/v2/channels/{channelId}")
    suspend fun getChannelDetails(@Path("channelId") channelId: String): Response<ChannelDetai
```

```
    @GET("api/v2/channels")
    suspend fun getChannels(
        @Query("type") type: String = "vtuber",
        @Query("org") organization: String,
        @Query("limit") limit: Int,
        @Query("offset") offset: Int,
        @Query("sort") sort: String = "suborg"
    ): Response<List<ChannelDetails>>

    @GET("api/v2/songs/hot")
    suspend fun getHotSongs(
        @Query("org") organization: String? = null,
        @Query("channel_id") channelId: String? = null
    ): Response<List<MusicdexSong>>

    @POST("api/v2/songs/latest")
    suspend fun getLatestSongs(@Body request: LatestSongsRequest): Response<PaginatedSongsResp



    @GET("/statics/orgs.json")
    suspend fun getOrganizations(): Response<List<Organization>>

    @GET("api/v2/users/favorites")
    suspend fun getFavoriteChannels(): Response<List<FavoriteChannelApiDto>>
}

// File: java\com\example\holodex\data\api\MusicdexApiService.kt
// File: java\com\example\holodex\data\api\MusicdexApiService.kt

package com.example.holodex.data.api

import com.example.holodex.data.model.discovery.DiscoveryResponse
import com.example.holodex.data.model.discovery.FullPlaylist
import com.example.holodex.data.model.discovery.MusicdexSong
import com.example.holodex.data.model.discovery.PlaylistStub
import retrofit2.Response
import retrofit2.http.Body
import retrofit2.http.GET
import retrofit2.http.POST
import retrofit2.http.Path
import retrofit2.http.Query

// --- Request/Response Models ---
data class PlaylistListResponse(
    val total: Int,
    val items: List<PlaylistStub>
)

data class ElasticsearchRequest(
    val q: String = "*",
    val query_by: String,
    val sort_by: String,
    val facet_by: String,
    val page: Int = 1,
```

```
    val per_page: Int = 25
    // Add other fields as needed for advanced search
)

data class ElasticsearchResponse<T>(
    val found: Int,
    val page: Int,
    val hits: List<Hit<T>>?
)

data class Hit<T>(
    val document: T
)

/**
 * Defines the public, non-authenticated endpoints for the music.holodex.net API.
 * Requires the X-APIKEY via an interceptor.
 */
interface MusicdexApiService {

    @GET("api/v2/musicdex/discovery/org/{org}")
    suspend fun getDiscoveryForOrg(@Path("org") organization: String): Response<DiscoveryRespo

    @GET("api/v2/musicdex/discovery/channel/{channelId}")
    suspend fun getDiscoveryForChannel(@Path("channelId") channelId: String): Response<Discove

    @GET("api/v2/musicdex/playlist/{playlistId}")
    suspend fun getPlaylistContent(@Path(value = "playlistId", encoded = true) playlistId: Str

    @GET("api/v2/musicdex/radio/{radioId}")
    suspend fun getRadioContent(
        @Path("radioId") radioId: String,
        @Query("offset") offset: Int = 0
    ): Response<FullPlaylist>

    @GET("api/v2/musicdex/discovery/org/{org}/playlists")
    suspend fun getOrgPlaylists(
        @Path("org") org: String,
        @Query("type") type: String,
        @Query("offset") offset: Int,
        @Query("limit") limit: Int
    ): Response<PlaylistListResponse>


    // --- NEW ENDPOINT for Goal 2.1 ---
    @POST("api/v2/musicdex/elasticsearch/search")
    suspend fun searchElasticsearch(
        @Body request: ElasticsearchRequest
    ): Response<ElasticsearchResponse<MusicdexSong>>
}

// File: java\com\example\holodex\data\api\PlaylistDto.kt
// File: java/com/example/holodex/data/api/PlaylistDto.kt (NEW FILE)
package com.example.holodex.data.api
```

```
import com.google.gson.annotations.SerializedName

/**
 * A dedicated Data Transfer Object (DTO) for deserializing playlist data from the Holodex API
 * The property names here EXACTLY match the JSON fields from the server.
 */
data class PlaylistDto(
    @SerializedName("id")
    val id: String,

    @SerializedName("title")
    val title: String?,

    @SerializedName("description")
    val description: String?,

    @SerializedName("owner")
    val owner: Long?,

    @SerializedName("type")
    val type: String?,

    @SerializedName("created_at")
    val createdAt: String?,

    @SerializedName("updated_at")
    val updatedAt: String?
)

// File: java\com\example\holodex\data\api\PlaylistRequestDtos.kt
// File: java/com/example/holodex/data/api/PlaylistRequestDtos.kt (NEW FILE)
package com.example.holodex.data.api

import com.google.gson.annotations.SerializedName

/**
 * A dedicated Data Transfer Object (DTO) for creating or updating a playlist via the API.
 * This class ONLY contains fields the server expects, solving the 500 error caused by
 * sending extra client-side fields like `is_deleted`.
 */
data class PlaylistUpdateRequest(
    @SerializedName("id")
    val id: String?, // Null when creating, non-null when updating

    @SerializedName("owner")
    val owner: Long,

    @SerializedName("title")
    val title: String?,

    @SerializedName("description")
    val description: String?,

    @SerializedName("type")
    val type: String = "ugp",
```

```
    /**
     * The complete list of song UUIDs. When sent, this will OVERWRITE the existing content.
     */
    @SerializedName("content")
    val content: List<String>
)

// File: java\com\example\holodex\data\cache\BrowseListCache.kt
package com.example.holodex.data.cache

import androidx.collection.LruCache
import com.example.holodex.data.db.BrowsePageDao
import com.example.holodex.data.db.CachedBrowsePage
import com.example.holodex.data.model.HolodexVideoItem
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.withContext
import timber.log.Timber
import java.util.concurrent.TimeUnit

class BrowseListCache(
    private val browsePageDao: BrowsePageDao,
    private val maxMemoryPages: Int = 5, // Max number of pages to keep in memory
    private val cacheValidityMs: Long = TimeUnit.HOURS.toMillis(1), // How long a cache entry
    private val staleValidityMs: Long = TimeUnit.DAYS.toMillis(1) // How long stale data can b
) {
    private data class CachePageEntry(
        val result: FetcherResult<HolodexVideoItem>, // Store HolodexVideoItem directly
        val timestamp: Long = System.currentTimeMillis()
    )

    // LruCache stores pages by their string key.
    private val memoryCache = object : LruCache<String, CachePageEntry>(maxMemoryPages) {
        override fun sizeOf(key: String, value: CachePageEntry): Int {
            // For LruCache, size is often 1 per entry if maxMemoryPages is the primary constr
            // If it were based on actual memory footprint, it'd be more complex.
            return 1 // Each entry counts as 1 towards maxMemoryPages
        }
    }

    private fun isFresh(timestamp: Long): Boolean {
        return System.currentTimeMillis() - timestamp < cacheValidityMs
    }

    private fun isStaleButAcceptable(timestamp: Long): Boolean {
        return System.currentTimeMillis() - timestamp < staleValidityMs
    }

    suspend fun get(key: BrowseCacheKey): FetcherResult<HolodexVideoItem>? = withContext(Dispa
        val stringKey = key.stringKey()
        memoryCache[stringKey]?.let { entry ->
            if (isFresh(entry.timestamp)) {
                Timber.d("BrowseListCache: Memory HIT (fresh) for key: $stringKey")
                return@withContext entry.result
            } else {
```

```
                Timber.d("BrowseListCache: Memory STALE for key: $stringKey, removing.")
                memoryCache.remove(stringKey) // Remove stale entry from memory
            }
        }

        // Check disk cache
        browsePageDao.getPage(stringKey)?.let { cachedPage ->
            if (isFresh(cachedPage.timestamp)) {
                Timber.d("BrowseListCache: Disk HIT (fresh) for key: $stringKey")
                val result = FetcherResult(cachedPage.data, cachedPage.totalAvailable)
                memoryCache.put(stringKey, CachePageEntry(result, cachedPage.timestamp)) // Po
                return@withContext result
            } else {
                // Data on disk is stale (older than cacheValidityMs) but not necessarily expi
                // We won't use it as "fresh" here. If network fails, getStale might pick it u
                Timber.d("BrowseListCache: Disk STALE for key: $stringKey. Will rely on networ
                // Optionally, one could delete it here if it's also older than staleValidityM
                // but cleanupExpired is a more global approach.
            }
        }
        Timber.d("BrowseListCache: Cache MISS for key: $stringKey")
        null
    }

    suspend fun getStale(key: BrowseCacheKey): FetcherResult<HolodexVideoItem>? = withContext(
        val stringKey = key.stringKey()
        // Try memory cache first (even if stale but acceptable)
        memoryCache[stringKey]?.let { entry ->
            if (isStaleButAcceptable(entry.timestamp)) {
                Timber.d("BrowseListCache: Memory HIT (stale but acceptable) for key: $stringK
                return@withContext entry.result
            }
        }

        // Try disk cache (stale but acceptable)
        browsePageDao.getPage(stringKey)?.let { cachedPage ->
            if (isStaleButAcceptable(cachedPage.timestamp)) {
                Timber.d("BrowseListCache: Disk HIT (stale but acceptable) for key: $stringKey
                // Note: Not re-populating memory cache with stale data from disk here,
                // as 'get' should be the one populating with fresh data.
                return@withContext FetcherResult(cachedPage.data, cachedPage.totalAvailable)
            } else {
                // Stale data on disk is too old even for fallback, remove it.
                Timber.d("BrowseListCache: Disk EXPIRED (too stale) for key: $stringKey, delet
                browsePageDao.deletePage(stringKey)
            }
        }
        Timber.d("BrowseListCache: Stale cache MISS for key: $stringKey")
        null
    }

    suspend fun store(key: BrowseCacheKey, result: FetcherResult<HolodexVideoItem>) = withCont
        val stringKey = key.stringKey()
        val currentTimestamp = System.currentTimeMillis()
        val entry = CachePageEntry(result, currentTimestamp)
```

```
        memoryCache.put(stringKey, entry)

        val cachedPage = CachedBrowsePage(
            pageKey = stringKey,
            data = result.data,
            totalAvailable = result.totalAvailable,
            timestamp = currentTimestamp
        )
        browsePageDao.insertPage(cachedPage)
        Timber.d("BrowseListCache: Stored data for key: $stringKey, items: ${result.data.size}
    }

    suspend fun invalidate(key: BrowseCacheKey) = withContext(Dispatchers.IO) {
        val stringKey = key.stringKey()
        memoryCache.remove(stringKey)
        browsePageDao.deletePage(stringKey)
        Timber.d("BrowseListCache: Invalidated data for key: $stringKey")
    }

    suspend fun clear() = withContext(Dispatchers.IO) {
        memoryCache.evictAll()
        browsePageDao.deleteAllBrowsePages()
        Timber.d("BrowseListCache: Cleared all browse cache data (memory and disk).")
    }

    suspend fun cleanupExpiredEntries() = withContext(Dispatchers.IO) {
        val tooOldTimestamp = System.currentTimeMillis() - staleValidityMs
        browsePageDao.deleteExpiredBrowsePages(tooOldTimestamp)
        // Memory cache entries are evicted by LRU or when found stale during 'get'.
        // Could also iterate memoryCache.snapshot().keys and remove based on timestamp if str
        Timber.d("BrowseListCache: Cleaned up expired disk entries older than $staleValidityMs
    }
}

// File: java\com\example\holodex\data\cache\CacheKey.kt
package com.example.holodex.data.cache

import com.example.holodex.viewmodel.state.BrowseFilterState
import com.google.gson.Gson

/**
 * Base interface for cache keys to ensure a string representation for Room.
 */
interface CacheKey {
    fun stringKey(): String
}

data class BrowseCacheKey(
    val filters: BrowseFilterState,
    val pageOffset: Int // Using offset as part of the key for paged data
) : CacheKey {
    override fun stringKey(): String {
        // A more stable serialization than default toString() for complex objects in keys.
        // Using Gson to serialize parts of the filter state.
        // Ensure BrowseFilterState and its members are GSON-serializable or use specific fiel
```

```
        val gson = Gson()
        val filterJson = gson.toJson(mapOf(
            "preset" to filters.selectedViewPreset.name,
            "org" to filters.selectedOrganization,
            "topic" to filters.selectedPrimaryTopic,
            "sortField" to filters.sortField.apiValue,
            "sortOrder" to filters.sortOrder.apiValue,
            "songSegmentFilter" to filters.songSegmentFilterMode.name,
            "status" to filters.status, // from selectedViewPreset
            "maxUpcomingHours" to filters.maxUpcomingHours // from selectedViewPreset
        ))
        return "browse_${filterJson}_offset=$pageOffset"
    }
}

data class SearchCacheKey(
    val query: String,
    val pageOffset: Int
) : CacheKey {
    override fun stringKey(): String {
        return "search_query=${query.trim().replace(" ", "_").take(100)}_offset=$pageOffset"
    }
}

// Add other keys as needed, e.g., for Favorites, LikedSegments, PlaylistItems if they use sim
// data class FavoritesCacheKey(val itemType: LikedItemType, val pageOffset: Int) : CacheKey {

// File: java\com\example\holodex\data\cache\CachePolicyAndException.kt
package com.example.holodex.data.cache

enum class CachePolicy {
    /**
     * Try to fetch from cache first.
     * If cache miss or expired, fetch from network.
     * If network fails, attempt to use stale cache.
     */
    CACHE_FIRST,

    /**
     * Try to fetch from network first.
     * If network fails, attempt to use cache (fresh or stale).
     */
    NETWORK_FIRST,

    /**
     * Only fetch from cache. Fails if not found or expired.
     * Might have a sub-policy to allow stale.
     */
    CACHE_ONLY,

    /**
     * Only fetch from network. Do not use cache.
     */
    NETWORK_ONLY
}
```

```
sealed class CacheException(message: String, cause: Throwable? = null) : Exception(message, ca
    class NotFound(message: String) : CacheException(message)
    class Expired(message: String) : CacheException(message)
    class StorageError(message: String, cause: Throwable) : CacheException(message, cause)
    class NetworkError(message: String, cause: Throwable?) : CacheException(message, cause)
}

// File: java\com\example\holodex\data\cache\FetcherResult.kt
// File: java/com/example/holodex/data/cache/FetcherResult.kt
package com.example.holodex.data.cache

data class FetcherResult<V>(
    val data: List<V>,
    val totalAvailable: Int?,
    val nextPageOffset: Int? = null,
    val nextPageCursor: Any? = null // Generic cursor for NewPipe's Page object
)

// File: java\com\example\holodex\data\cache\SearchListCache.kt
package com.example.holodex.data.cache

import androidx.collection.LruCache
import com.example.holodex.data.db.CachedSearchPage
import com.example.holodex.data.db.SearchPageDao
import com.example.holodex.data.model.HolodexVideoItem
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.withContext
import timber.log.Timber
import java.util.concurrent.TimeUnit

class SearchListCache(
    private val searchPageDao: SearchPageDao,
    private val maxMemoryPages: Int = 3, // Search might be less frequently revisited per spec
    private val cacheValidityMs: Long = TimeUnit.MINUTES.toMillis(30),
    private val staleValidityMs: Long = TimeUnit.HOURS.toMillis(12)
) {
    private data class CachePageEntry(
        val result: FetcherResult<HolodexVideoItem>,
        val timestamp: Long = System.currentTimeMillis()
    )

    private val memoryCache = object : LruCache<String, CachePageEntry>(maxMemoryPages) {
        override fun sizeOf(key: String, value: CachePageEntry): Int = 1
    }

    private fun isFresh(timestamp: Long): Boolean = System.currentTimeMillis() - timestamp < c
    private fun isStaleButAcceptable(timestamp: Long): Boolean = System.currentTimeMillis() -

    suspend fun get(key: SearchCacheKey): FetcherResult<HolodexVideoItem>? = withContext(Dispa
        val stringKey = key.stringKey()
        memoryCache[stringKey]?.let { entry ->
            if (isFresh(entry.timestamp)) {
                Timber.d("SearchListCache: Memory HIT (fresh) for key: $stringKey")
                return@withContext entry.result
```

```
        } else {
            Timber.d("SearchListCache: Memory STALE for key: $stringKey, removing.")
            memoryCache.remove(stringKey)
        }
    }

    searchPageDao.getPage(stringKey)?.let { cachedPage ->
        if (isFresh(cachedPage.timestamp)) {
            Timber.d("SearchListCache: Disk HIT (fresh) for key: $stringKey")
            val result = FetcherResult(cachedPage.data, cachedPage.totalAvailable)
            memoryCache.put(stringKey, CachePageEntry(result, cachedPage.timestamp))
            return@withContext result
        }
    }
    Timber.d("SearchListCache: Cache MISS for key: $stringKey")
    null
}

suspend fun getStale(key: SearchCacheKey): FetcherResult<HolodexVideoItem>? = withContext(
    val stringKey = key.stringKey()
    memoryCache[stringKey]?.let { entry ->
        if (isStaleButAcceptable(entry.timestamp)) {
            Timber.d("SearchListCache: Memory HIT (stale acceptable) for key: $stringKey")
            return@withContext entry.result
        }
    }

    searchPageDao.getPage(stringKey)?.let { cachedPage ->
        if (isStaleButAcceptable(cachedPage.timestamp)) {
            Timber.d("SearchListCache: Disk HIT (stale acceptable) for key: $stringKey")
            return@withContext FetcherResult(cachedPage.data, cachedPage.totalAvailable)
        } else {
            Timber.d("SearchListCache: Disk EXPIRED (too stale) for key: $stringKey, delet
            searchPageDao.deletePage(stringKey)
        }
    }
    Timber.d("SearchListCache: Stale cache MISS for key: $stringKey")
    null
}

suspend fun store(key: SearchCacheKey, result: FetcherResult<HolodexVideoItem>) = withCont
    val stringKey = key.stringKey()
    val currentTimestamp = System.currentTimeMillis()
    val entry = CachePageEntry(result, currentTimestamp)
    memoryCache.put(stringKey, entry)

    val cachedPage = CachedSearchPage(
        pageKey = stringKey,
        data = result.data,
        totalAvailable = result.totalAvailable,
        timestamp = currentTimestamp
    )
    searchPageDao.insertPage(cachedPage)
    Timber.d("SearchListCache: Stored data for key: $stringKey, items: ${result.data.size}
}
```

```
    suspend fun invalidate(key: SearchCacheKey) = withContext(Dispatchers.IO) {
        val stringKey = key.stringKey()
        memoryCache.remove(stringKey)
        searchPageDao.deletePage(stringKey)
        Timber.d("SearchListCache: Invalidated data for key: $stringKey")
    }

    suspend fun clear() = withContext(Dispatchers.IO) {
        memoryCache.evictAll()
        searchPageDao.deleteAllSearchPages()
        Timber.d("SearchListCache: Cleared all search cache data.")
    }

    suspend fun cleanupExpiredEntries() = withContext(Dispatchers.IO) {
        val tooOldTimestamp = System.currentTimeMillis() - staleValidityMs
        searchPageDao.deleteExpiredSearchPages(tooOldTimestamp)
        Timber.d("SearchListCache: Cleaned up expired disk entries older than $staleValidityMs
    }
}

// File: java\com\example\holodex\data\db\AppDatabase.kt
// File: java/com/example/holodex/data/db/AppDatabase.kt
package com.example.holodex.data.db

import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase
import androidx.room.TypeConverter
import androidx.room.TypeConverters
import androidx.room.migration.Migration
import androidx.sqlite.db.SupportSQLiteDatabase
import com.example.holodex.playback.data.model.PersistedPlaybackItemEntity
import com.example.holodex.playback.data.model.PersistedPlaybackStateDao
import com.example.holodex.playback.data.model.PersistedPlaybackStateEntity

@Database(
    entities = [
        CachedVideoEntity::class,
        CachedSongEntity::class,
        PersistedPlaybackItemEntity::class,
        PersistedPlaybackStateEntity::class,
        LikedItemEntity::class,
        PlaylistEntity::class,
        PlaylistItemEntity::class,
        CachedBrowsePage::class,
        CachedSearchPage::class,
        DownloadedItemEntity::class,
        ParentVideoMetadataEntity::class,
        HistoryItemEntity::class,
        CachedDiscoveryResponse::class,
        FavoriteChannelEntity::class,
        SyncMetadataEntity::class,
        LocalFavoriteEntity::class,
```

```
        ExternalChannelEntity::class,
        StarredPlaylistEntity::class,
        LocalPlaylistEntity::class,
        LocalPlaylistItemEntity::class
    ],
    version = 21,
    exportSchema = true
)
@TypeConverters(
    HolodexSongListConverter::class,
    LikedItemTypeConverter::class,
    HolodexVideoItemListConverter::class,
    StringListConverter::class,
    DiscoveryResponseConverter::class,
    DownloadStatusConverter::class,
    SyncStatusConverter::class
)
abstract class AppDatabase : RoomDatabase() {
    abstract fun videoDao(): VideoDao
    abstract fun persistedPlaybackStateDao(): PersistedPlaybackStateDao
    abstract fun likedItemDao(): LikedItemDao
    abstract fun playlistDao(): PlaylistDao
    abstract fun browsePageDao(): BrowsePageDao
    abstract fun searchPageDao(): SearchPageDao
    abstract fun downloadedItemDao(): DownloadedItemDao
    abstract fun parentVideoMetadataDao(): ParentVideoMetadataDao
    abstract fun historyDao(): HistoryDao
    abstract fun favoriteChannelDao(): FavoriteChannelDao
    abstract fun discoveryDao(): DiscoveryDao
    abstract fun syncMetadataDao(): SyncMetadataDao
    abstract fun starredPlaylistDao(): StarredPlaylistDao
    abstract fun localDao(): LocalDao

    companion object {
        @Volatile
        private var INSTANCE: AppDatabase? = null


        val MIGRATION_18_19: Migration = object : Migration(18, 19) {
            override fun migrate(db: SupportSQLiteDatabase) {
                // No operation needed. The schema for version 18 is identical to the
                // final schema for version 19. This migration simply tells Room
                // that the transition is safe and requires no changes.
            }
        }
        val MIGRATION_19_20: Migration = object : Migration(19, 20) {
            override fun migrate(db: SupportSQLiteDatabase) {
                db.execSQL("ALTER TABLE `playlist_items` ADD COLUMN `is_local_only` INTEGER NO

                db.execSQL("""
                    CREATE TABLE IF NOT EXISTS `local_favorites` (
                        `itemId` TEXT NOT NULL,
                        `videoId` TEXT NOT NULL,
                        `channelId` TEXT NOT NULL,
                        `title` TEXT NOT NULL,
```

```
                `artistText` TEXT NOT NULL,
                `artworkUrl` TEXT,
                `durationSec` INTEGER NOT NULL,
                `isSegment` INTEGER NOT NULL,
                `songStartSec` INTEGER,
                `songEndSec` INTEGER,
                PRIMARY KEY(`itemId`)
            )
        """)

        db.execSQL("""
            CREATE TABLE IF NOT EXISTS `external_channels` (
                `channelId` TEXT NOT NULL,
                `name` TEXT NOT NULL,
                `photoUrl` TEXT,
                `lastCheckedTimestamp` INTEGER NOT NULL,
                `status` TEXT NOT NULL,
                `errorCount` INTEGER NOT NULL,
                PRIMARY KEY(`channelId`)
            )
        """)

        // *** THE FIX IS HERE ***
        db.execSQL("""
            CREATE TABLE IF NOT EXISTS `local_playlists` (
                `localPlaylistId` INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
                `name` TEXT NOT NULL,
                `description` TEXT,
                `createdAt` INTEGER NOT NULL
            )
        """)
        // *** END OF FIX ***

        db.execSQL("""
            CREATE TABLE IF NOT EXISTS `local_playlist_items` (
                `playlistOwnerId` INTEGER NOT NULL,
                `itemId` TEXT NOT NULL,
                `videoId` TEXT NOT NULL,
                `itemOrder` INTEGER NOT NULL,
                `title` TEXT NOT NULL,
                `artistText` TEXT NOT NULL,
                `artworkUrl` TEXT,
                `durationSec` INTEGER NOT NULL,
                `channelId` TEXT NOT NULL,
                `isSegment` INTEGER NOT NULL,
                `songStartSec` INTEGER,
                `songEndSec` INTEGER,
                PRIMARY KEY(`playlistOwnerId`, `itemId`)
            )
        """)
    }
}
val MIGRATION_20_21: Migration = object : Migration(20, 21) {
    override fun migrate(db: SupportSQLiteDatabase) {
        // --- FIX FOR local_playlists TABLE ---
```

```
            db.execSQL("ALTER TABLE `local_playlists` RENAME TO `local_playlists_old`")
            db.execSQL("""
                CREATE TABLE `local_playlists` (
                    `localPlaylistId` INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
                    `name` TEXT NOT NULL,
                    `description` TEXT,
                    `createdAt` INTEGER NOT NULL
                )
            """)
            // Copy data, providing a default for the new column.
            // Note: We use System.currentTimeMillis() for createdAt to give old data a ti
            db.execSQL("""
                INSERT INTO `local_playlists` (localPlaylistId, name, description, created
                SELECT playlistId, name, description, ${System.currentTimeMillis()} FROM `
            """)
            db.execSQL("DROP TABLE `local_playlists_old`")

            // --- NEW FIX FOR local_playlist_items TABLE ---
            // This table was also created incorrectly in the 19->20 migration. We fix it
            db.execSQL("DROP TABLE IF EXISTS `local_playlist_items`") // It contains no us
            db.execSQL("""
                CREATE TABLE IF NOT EXISTS `local_playlist_items` (
                    `playlistOwnerId` INTEGER NOT NULL,
                    `itemId` TEXT NOT NULL,
                    `videoId` TEXT NOT NULL,
                    `itemOrder` INTEGER NOT NULL,
                    `title` TEXT NOT NULL,
                    `artistText` TEXT NOT NULL,
                    `artworkUrl` TEXT,
                    `durationSec` INTEGER NOT NULL,
                    `channelId` TEXT NOT NULL,
                    `isSegment` INTEGER NOT NULL,
                    `songStartSec` INTEGER,
                    `songEndSec` INTEGER,
                    PRIMARY KEY(`playlistOwnerId`, `itemId`)
                )
            """)
        }
    }
    fun getDatabase(context: Context): AppDatabase {
        return INSTANCE ?: synchronized(this) {
            val instance = Room.databaseBuilder(
                context.applicationContext,
                AppDatabase::class.java,
                "holodex_music_app_database"
            )
                .addMigrations(MIGRATION_18_19,
                    MIGRATION_19_20,
                    MIGRATION_20_21
                )
                .build()
            INSTANCE = instance
            instance
        }
    }
```

```
    }
}


class DownloadStatusConverter {
    @TypeConverter
    fun fromDownloadStatus(value: DownloadStatus?): String? {
        return value?.name
    }

    @TypeConverter
    fun toDownloadStatus(value: String?): DownloadStatus? {
        return value?.let {
            try {
                DownloadStatus.valueOf(it)
            } catch (_: IllegalArgumentException) {
                null
            }
        }
    }
}

class LikedItemTypeConverter {
    @TypeConverter
    fun fromLikedItemType(value: LikedItemType?): String? {
        return value?.name
    }

    @TypeConverter
    fun toLikedItemType(value: String?): LikedItemType? {
        return value?.let {
            try {
                LikedItemType.valueOf(it)
            } catch (_: IllegalArgumentException) {
                null
            }
        }
    }
}
class SyncStatusConverter {
    @TypeConverter
    fun fromSyncStatus(value: SyncStatus?): String? {
        return value?.name
    }

    @TypeConverter
    fun toSyncStatus(value: String?): SyncStatus? {
        return value?.let {
            try {
                SyncStatus.valueOf(it)
            } catch (_: IllegalArgumentException) {
                null
            }
        }
```

```
    }
}


// File: java\com\example\holodex\data\db\BrowsePageDao.kt
package com.example.holodex.data.db

import androidx.room.Dao
import androidx.room.Insert
import androidx.room.OnConflictStrategy
import androidx.room.Query

@Dao
interface BrowsePageDao {
    @Query("SELECT * FROM cached_browse_pages WHERE pageKey = :pageKey")
    suspend fun getPage(pageKey: String): CachedBrowsePage?

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertPage(page: CachedBrowsePage)

    @Query("DELETE FROM cached_browse_pages WHERE pageKey = :pageKey")
    suspend fun deletePage(pageKey: String)

    /**
     * Deletes pages older than the given timestamp for browse caches.
     * The pageKey for browse pages might start with a common prefix like "browse_".
     */
    @Query("DELETE FROM cached_browse_pages WHERE timestamp < :expiredTime")
    suspend fun deleteExpiredBrowsePages(expiredTime: Long)

    /**
     * Deletes all pages from the browse cache.
     */
    @Query("DELETE FROM cached_browse_pages")
    suspend fun deleteAllBrowsePages()

    // Optional: Method to get cache size for monitoring
    @Query("SELECT COUNT(pageKey) FROM cached_browse_pages")
    suspend fun getBrowseCacheSize(): Int
}

// File: java\com\example\holodex\data\db\CachedPageEntities.kt
package com.example.holodex.data.db

import androidx.room.ColumnInfo
import androidx.room.Entity
import androidx.room.Index
import com.example.holodex.data.model.HolodexVideoItem // Ensure this import is correct
import com.example.holodex.data.model.discovery.DiscoveryResponse

/**
 * Entity to store a "page" of browsed video items.
 * The pageKey should uniquely identify the filter set and offset/page number.
 */
@Entity(
```

```
    tableName = "cached_browse_pages",
    primaryKeys = ["pageKey"],
    indices = [Index(value = ["timestamp"])]
)
data class CachedBrowsePage(
    val pageKey: String, // Example: "browse_preset=LATEST_STREAMS_org=Hololive_topic=singing_
    @ColumnInfo(name = "data_list")
    val data: List<HolodexVideoItem>, // Will use TypeConverter
    val timestamp: Long = System.currentTimeMillis(),
    val totalAvailable: Int? = null // Total items API reported for this query, to help determ
)

/**
 * Entity to store a "page" of searched video items.
 * The pageKey should uniquely identify the search query and offset/page number.
 */
@Entity(
    tableName = "cached_search_pages",
    primaryKeys = ["pageKey"],
    indices = [Index(value = ["timestamp"])]
)
data class CachedSearchPage(
    val pageKey: String, // Example: "search_query=my_search_term_offset=0"
    @ColumnInfo(name = "data_list")
    val data: List<HolodexVideoItem>, // Will use TypeConverter
    val timestamp: Long = System.currentTimeMillis(),
    val totalAvailable: Int? = null
)

@Entity(
    tableName = "cached_discovery_responses",
    primaryKeys = ["pageKey"],
    indices = [Index(value = ["timestamp"])]
)
data class CachedDiscoveryResponse(
    val pageKey: String, // e.g., "discovery_org_Hololive", "discovery_favorites"
    @ColumnInfo(name = "data_response")
    val data: DiscoveryResponse, // Will use TypeConverter
    val timestamp: Long = System.currentTimeMillis()
)



// File: java\com\example\holodex\data\db\Converters.kt
// File: java/com/example/holodex/data/db/Converters.kt
package com.example.holodex.data.db

import androidx.room.TypeConverter
import com.example.holodex.data.model.HolodexVideoItem
import com.example.holodex.data.model.discovery.DiscoveryResponse
import com.google.gson.Gson
import com.google.gson.reflect.TypeToken

class HolodexVideoItemListConverter {
    private val gson = Gson()
```

```
    @TypeConverter
    fun fromVideoItemList(value: List<HolodexVideoItem>?): String? {
        return value?.let { gson.toJson(it) }
    }

    @TypeConverter
    fun toVideoItemList(value: String?): List<HolodexVideoItem>? {
        return value?.let {
            try {
                val listType = object : TypeToken<List<HolodexVideoItem>>() {}.type
                gson.fromJson(it, listType)
            } catch (e: Exception) {
                // Handle potential GSON parsing errors, e.g., if JSON is malformed
                null // Or throw, or log
            }
        }
    }
}

// Adding the StringListConverter here as well for co-location
class StringListConverter {
    @TypeConverter
    fun fromStringList(value: List<String>?): String? {
        return value?.joinToString(separator = "|||")
    }

    @TypeConverter
    fun toStringList(value: String?): List<String>? {
        return value?.split("|||")?.filter { it.isNotEmpty() }
    }
}
class DiscoveryResponseConverter {
    private val gson = Gson()

    @TypeConverter
    fun fromDiscoveryResponse(value: DiscoveryResponse?): String? {
        return value?.let { gson.toJson(it) }
    }

    @TypeConverter
    fun toDiscoveryResponse(value: String?): DiscoveryResponse? {
        return value?.let {
            try {
                gson.fromJson(it, DiscoveryResponse::class.java)
            } catch (e: Exception) {
                null
            }
        }
    }
}


// File: java\com\example\holodex\data\db\DiscoveryDao.kt
package com.example.holodex.data.db
```

```
import androidx.room.Dao
import androidx.room.Insert
import androidx.room.OnConflictStrategy
import androidx.room.Query

@Dao
interface DiscoveryDao {
    @Query("SELECT * FROM cached_discovery_responses WHERE pageKey = :pageKey")
    suspend fun getResponse(pageKey: String): CachedDiscoveryResponse?

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertResponse(response: CachedDiscoveryResponse)

    @Query("DELETE FROM cached_discovery_responses WHERE timestamp < :expiredTime")
    suspend fun deleteExpiredResponses(expiredTime: Long)
}

// File: java\com\example\holodex\data\db\DownloadedItemDao.kt
package com.example.holodex.data.db

import androidx.room.Dao
import androidx.room.Insert
import androidx.room.OnConflictStrategy
import androidx.room.Query
import kotlinx.coroutines.flow.Flow

@Dao
interface DownloadedItemDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertOrUpdate(item: DownloadedItemEntity)

    @Query("SELECT * FROM downloaded_items WHERE videoId = :videoId LIMIT 1")
    suspend fun getById(videoId: String): DownloadedItemEntity?

    @Query("SELECT * FROM downloaded_items WHERE videoId = :videoId LIMIT 1")
    fun getDownloadByIdFlow(videoId: String): Flow<DownloadedItemEntity?>

    @Query("SELECT * FROM downloaded_items ORDER BY downloadedAt DESC")
    fun getAllDownloads(): Flow<List<DownloadedItemEntity>>

    @Query("SELECT * FROM downloaded_items WHERE videoId IN (:ids)")
    suspend fun getByIds(ids: List<String>): List<DownloadedItemEntity>

    @Query("DELETE FROM downloaded_items WHERE videoId = :videoId")
    suspend fun deleteById(videoId: String)

    @Query("UPDATE downloaded_items SET downloadStatus = :status, localFileUri = :uri, fileNam
    suspend fun updateStatusToCompleted(
        videoId: String,
        status: DownloadStatus = DownloadStatus.COMPLETED,
        uri: String,
        fileName: String,
        timestamp: Long
    )
```

```
    @Query("UPDATE downloaded_items SET downloadStatus = :status WHERE videoId = :videoId")
    suspend fun updateStatus(videoId: String, status: DownloadStatus)

    @Query("UPDATE downloaded_items SET progress = :progress WHERE videoId = :videoId")
    suspend fun updateProgress(videoId: String, progress: Int)

    // FIX: Change parameter name from `itemId` to `videoId` for consistency with the query co
    // This is a minor readability improvement, not a functional change.
    @Query("UPDATE downloaded_items SET progress = :progress, downloadStatus = :status WHERE v
    suspend fun updateProgressAndStatus(videoId: String, progress: Int, status: DownloadStatus

    // FIX: Change parameter name from `itemId` to `videoId` for consistency.
    @Query("UPDATE downloaded_items SET localFileUri = :uri WHERE videoId = :videoId")
    suspend fun updateLocalFileUri(videoId: String, uri: String)

    // FIX: Change parameter name from `itemId` to `videoId` for consistency.
    @Query("UPDATE downloaded_items SET downloadedAt = :timestamp WHERE videoId = :videoId")
    suspend fun updateDownloadedAt(videoId: String, timestamp: Long)
}

// File: java\com\example\holodex\data\db\entities.kt
// File: java/com/example/holodex/data/db/entities.kt
package com.example.holodex.data.db

import androidx.room.ColumnInfo
import androidx.room.Embedded
import androidx.room.Entity
import androidx.room.ForeignKey
import androidx.room.Index
import androidx.room.PrimaryKey
import androidx.room.Relation
import com.example.holodex.data.model.HolodexChannelMin
import com.example.holodex.data.model.HolodexSong
import com.example.holodex.data.model.HolodexVideoItem
import timber.log.Timber
enum class SyncStatus {
    /** The item's state is consistent with the server's. */
    SYNCED,
    /** The item was created or modified locally and needs to be uploaded. */
    DIRTY,
    /** The item was deleted locally and needs to be deleted from the server. */
    PENDING_DELETE,

}

@Entity(tableName = "videos")
data class CachedVideoEntity(
    @PrimaryKey val id: String,
    val title: String,
    val type: String,
    @ColumnInfo(name = "topic_id") val topicId: String?,
    @ColumnInfo(name = "published_at") val publishedAt: String?,
    @ColumnInfo(name = "available_at") val availableAt: String,
    val duration: Long,
    val status: String,
```

```
    @ColumnInfo(name = "song_count") val songCount: Int?,
    val description: String?, // This field will store the potentially truncated description
    @Embedded(prefix = "channel_")
    val channel: HolodexChannelMin,
    @ColumnInfo(
        name = "fetched_at_ms",
        defaultValue = "0"
    ) val fetchedAtMs: Long = System.currentTimeMillis(),
    @ColumnInfo(name = "list_query_key") val listQueryKey: String? = null,
    @ColumnInfo(
        name = "insertion_order",
        defaultValue = "0"
    ) val insertionOrder: Int = 0 // NEW FIELD
)

@Entity(
    tableName = "songs",
    primaryKeys = ["video_id", "start_time_seconds"],
    foreignKeys = [ForeignKey(
        entity = CachedVideoEntity::class,
        parentColumns = ["id"],
        childColumns = ["video_id"],
        onDelete = ForeignKey.CASCADE
    )],
    indices = [Index(value = ["video_id"])]
)
data class CachedSongEntity(
    @ColumnInfo(name = "video_id") val videoId: String,
    val name: String,
    @ColumnInfo(name = "start_time_seconds") val startTimeSeconds: Int,
    @ColumnInfo(name = "end_time_seconds") val endTimeSeconds: Int,
    @ColumnInfo(name = "original_artist") val originalArtist: String?,
    @ColumnInfo(name = "itunes_id") val itunesId: Int?,
    @ColumnInfo(name = "art_url") val artUrl: String?
)


// --- Mappers ---

fun HolodexVideoItem.toEntity(
    queryKey: String? = null,
    currentTimestamp: Long = System.currentTimeMillis(),
    orgNameFromRequest: String? = null,
    insertionOrder: Int = 0 // NEW PARAMETER
): CachedVideoEntity {
    val effectiveOrg = orgNameFromRequest ?: this.channel.org

    // Truncate description to a reasonable length
    val maxDescriptionLength = 1000 // Max 1000 characters for DB storage
    var truncatedDescription = this.description
    if (this.description?.length ?: 0 > maxDescriptionLength) {
        // Ensure truncation doesn't break multi-byte characters if present, though it's less
        truncatedDescription = this.description!!.substring(0, maxDescriptionLength) + "..."
        Timber.tag("HoloVideoItem.toEntity")
            .w("Description for video ${this.id} was truncated from ${this.description.length}
```

```
    }

    return CachedVideoEntity(
        id = this.id,
        title = this.title,
        type = this.type,
        topicId = this.topicId,
        publishedAt = this.publishedAt,
        availableAt = this.availableAt,
        duration = this.duration,
        status = this.status,
        songCount = this.songcount,
        description = truncatedDescription, // Use the (potentially) truncated description
        channel = HolodexChannelMin(
            id = this.channel.id,
            name = this.channel.name,
            englishName = this.channel.englishName,
            photoUrl = this.channel.photoUrl,
            type = this.channel.type,
            org = effectiveOrg
        ),
        fetchedAtMs = currentTimestamp,
        listQueryKey = queryKey,
        insertionOrder = insertionOrder // Populate new field
    )
}

fun HolodexSong.toEntity(videoIdParam: String): CachedSongEntity {
    return CachedSongEntity(
        videoId = videoIdParam,
        name = this.name,
        startTimeSeconds = this.start,
        endTimeSeconds = this.end,
        originalArtist = null, // This field was not in HolodexSong, keep as null or populate
        itunesId = this.itunesId,
        artUrl = this.artUrl
    )
}

fun CachedVideoEntity.toDomain(songsList: List<HolodexSong>? = null): HolodexVideoItem {
    return HolodexVideoItem(
        id = this.id,
        title = this.title,
        type = this.type,
        topicId = this.topicId,
        publishedAt = this.publishedAt,
        availableAt = this.availableAt,
        duration = this.duration,
        status = this.status,
        songcount = this.songCount,
        description = this.description, // This will be the (potentially) truncated descriptio
        channel = this.channel,
        songs = songsList
    )
}
```

```
fun CachedSongEntity.toDomain(): HolodexSong {
    return HolodexSong(
        name = this.name,
        start = this.startTimeSeconds,
        end = this.endTimeSeconds,
        itunesId = this.itunesId,
        videoId = this.videoId, // Ensure this is mapped back
        artUrl = this.artUrl
    )
}

data class VideoWithSongs(
    @Embedded val video: CachedVideoEntity,
    @Relation(
        parentColumn = "id",
        entityColumn = "video_id"
    )
    val songs: List<CachedSongEntity>
) {
    fun toDomain(): HolodexVideoItem {
        return video.toDomain(songsList = songs.map { it.toDomain() })
    }
}


enum class DownloadStatus {
    NOT_DOWNLOADED,
    ENQUEUED,
    DOWNLOADING,
    COMPLETED,
    FAILED,
    PROCESSING,
    EXPORT_FAILED,
    PAUSED,
    DELETING
}

@Entity(tableName = "downloaded_items")
data class DownloadedItemEntity(
    @PrimaryKey val videoId: String,

    // Metadata snapshot
    val title: String,
    val artistText: String,
    val channelId: String,
    val artworkUrl: String?,
    val durationSec: Long,

    // Download-specific info
    val localFileUri: String?,
    val downloadStatus: DownloadStatus,
    val downloadedAt: Long?,

    val fileName: String,
```

```
    val targetFormat: String, // Will store "M4A" or "OGG"

    // WorkManager's ID is a UUID string
    val downloadId: String?,

    // Progress field is useful for the UI
    val progress: Int = 0,
    @ColumnInfo(name = "track_number")
    val trackNumber: Int? = null
)

enum class LikedItemType {
    VIDEO,
    SONG_SEGMENT
}

@Entity(tableName = "liked_items")
data class LikedItemEntity(
    @PrimaryKey val itemId: String,
    val videoId: String,
    @ColumnInfo(name = "item_type", defaultValue = "VIDEO")
    val itemType: LikedItemType,
    @ColumnInfo(name = "server_id")
    val serverId: String?, // The UUID from the server
    // General snapshot fields (populated for both types)
    @ColumnInfo(name = "title_snapshot") val titleSnapshot: String,
    @ColumnInfo(name = "artist_text_snapshot") val artistTextSnapshot: String,
    @ColumnInfo(name = "album_text_snapshot") val albumTextSnapshot: String?,
    @ColumnInfo(name = "artwork_url_snapshot") val artworkUrlSnapshot: String?,
    @ColumnInfo(name = "description_snapshot") val descriptionSnapshot: String?,
    @ColumnInfo(name = "channel_id_snapshot") val channelIdSnapshot: String,
    @ColumnInfo(name = "duration_sec_snapshot") val durationSecSnapshot: Long,

    // Fields specific to SONG_SEGMENT (nullable if itemType is VIDEO)
    @ColumnInfo(name = "actual_song_name") val actualSongName: String? = null,      // Explici
    @ColumnInfo(name = "actual_song_artist") val actualSongArtist: String? = null, // E.g., Or
    @ColumnInfo(name = "actual_song_artwork_url") val actualSongArtworkUrl: String? = null, //

    @ColumnInfo(name = "song_start_seconds") val songStartSeconds: Int? = null,
    @ColumnInfo(name = "song_end_seconds") val songEndSeconds: Int? = null,

    @ColumnInfo(name = "liked_at") val likedAt: Long = System.currentTimeMillis(),
    @ColumnInfo(name = "last_modified_at", defaultValue = "0")
    val lastModifiedAt: Long = System.currentTimeMillis(),

    @ColumnInfo(name = "sync_status", defaultValue = "'DIRTY'")
    var syncStatus: SyncStatus = SyncStatus.DIRTY
) {
    companion object {
        fun generateVideoItemId(videoId: String): String = videoId
        fun generateSongItemId(videoId: String, songStartSeconds: Int): String =
            "${videoId}_${songStartSeconds}"
    }
}
```

```
// --- START REPLACEMENT ---
/**
 * Represents a user-created playlist in the local Room database.
 * This class is now decoupled from the network layer and contains only the fields
 * necessary for persistence and client-side sync logic.
 */
@Entity(
    tableName = "playlists",
    indices = [Index(value = ["server_id"])]
)
data class PlaylistEntity(
    @PrimaryKey(autoGenerate = true)
    var playlistId: Long = 0, // Local DB primary key, auto-generated

    // Server-side fields
    @ColumnInfo(name = "server_id")
    var serverId: String?, // The UUID from the server, nullable for local-only playlists

    var name: String?,
    var description: String?,
    var owner: Long?,
    var type: String = "ugp",

    @ColumnInfo(name = "created_at")
    var createdAt: String?,

    @ColumnInfo(name = "updated_at")
    var last_modified_at: String?, // Cleaned up property name to match column

    // Client-side sync state fields, ignored by network serialization
    @ColumnInfo(name = "is_deleted", defaultValue = "0")
    var isDeleted: Boolean = false,

    @ColumnInfo(name = "sync_status", defaultValue = "'DIRTY'")
    var syncStatus: SyncStatus = SyncStatus.DIRTY
)
// --- END REPLACEMENT ---


@Entity(
    tableName = "playlist_items",
    primaryKeys = ["playlist_owner_id", "item_id_in_playlist"],
    foreignKeys = [
        ForeignKey(
            entity = PlaylistEntity::class,
            parentColumns = ["playlistId"],
            childColumns = ["playlist_owner_id"],
            onDelete = ForeignKey.CASCADE
        )

    ],
    indices = [Index(value = ["playlist_owner_id"])]
)
data class PlaylistItemEntity(
```

```
    @ColumnInfo(name = "playlist_owner_id") val playlistOwnerId: Long,
    @ColumnInfo(name = "item_id_in_playlist") val itemIdInPlaylist: String, // Mirrors structu
    @ColumnInfo(name = "video_id_for_item") val videoIdForItem: String, // Parent video ID
    @ColumnInfo(name = "item_type_in_playlist") val itemTypeInPlaylist: LikedItemType,
    @ColumnInfo(name = "is_local_only", defaultValue = "0") val isLocalOnly: Boolean = false,
    // Optional: Snapshot of data for quick display, similar to LikedItemEntity
    @ColumnInfo(name = "song_start_seconds_playlist") val songStartSecondsPlaylist: Int? = nul
    @ColumnInfo(name = "song_end_seconds_playlist") val songEndSecondsPlaylist: Int? = null,
    @ColumnInfo(name = "song_name_playlist") val songNamePlaylist: String? = null,
    @ColumnInfo(name = "song_artist_text_playlist") val songArtistTextPlaylist: String? = null
    @ColumnInfo(name = "song_artwork_url_playlist") val songArtworkUrlPlaylist: String? = null
    // Could also store video title snapshot if it's a video item

    @ColumnInfo(name = "added_at", defaultValue = "CURRENT_TIMESTAMP")
    val addedAt: Long = System.currentTimeMillis(),
    @ColumnInfo(name = "item_order") val itemOrder: Int, // Order of the item within this spec
    @ColumnInfo(name = "last_modified_at", defaultValue = "0")
    val lastModifiedAt: Long = System.currentTimeMillis(),

    @ColumnInfo(name = "sync_status", defaultValue = "'DIRTY'")
    var syncStatus: SyncStatus = SyncStatus.DIRTY
)

@Entity(tableName = "history_items")
data class HistoryItemEntity(
    // The primary key is when the item was played.
    @PrimaryKey
    val playedAtTimestamp: Long,

    // Core identifiers for local use.
    @ColumnInfo(index = true)
    val itemId: String, // Our composite key: "videoId_startTime"
    val videoId: String,
    val songStartSeconds: Int,

    // Snapshot of metadata for fast UI display.
    val title: String,
    val artistText: String,
    val artworkUrl: String?,
    val durationSec: Long,
    val channelId: String

)

@Entity(tableName = "favorite_channels")
data class FavoriteChannelEntity(
    @PrimaryKey
    val id: String,
    val name: String?,
    val englishName: String?,
    @ColumnInfo(name = "photo")
    val photoUrl: String?,
    val org: String?,
    val subscriberCount: Int?,
    val twitter: String?,
```

```
    @ColumnInfo(name = "favorited_at_timestamp", defaultValue = "0")
    val favoritedAtTimestamp: Long = System.currentTimeMillis(),
    @ColumnInfo(name = "is_deleted", defaultValue = "0")
    val isDeleted: Boolean = false,
    @ColumnInfo(name = "sync_status", defaultValue = "'DIRTY'")
    val syncStatus: SyncStatus = SyncStatus.DIRTY
)
@Entity(tableName = "sync_metadata")
data class SyncMetadataEntity(
    @PrimaryKey val dataType: String, // e.g., "likes", "history"
    val lastSyncTimestamp: Long
)
@Entity(tableName = "parent_video_metadata")
data class ParentVideoMetadataEntity(
    @PrimaryKey val videoId: String,
    val title: String,
    val channelName: String,
    val channelId: String,
    val thumbnailUrl: String?,
    val description: String?,
    val totalDurationSec: Long
)

// File: java\com\example\holodex\data\db\FavoriteChannelDao.kt
// File: java/com/example/holodex/data/db/FavoriteChannelDao.kt

package com.example.holodex.data.db

import androidx.room.Dao
import androidx.room.Insert
import androidx.room.OnConflictStrategy
import androidx.room.Query
import kotlinx.coroutines.flow.Flow

@Dao
interface FavoriteChannelDao {

    @Query("SELECT * FROM favorite_channels WHERE is_deleted = 0 ORDER BY favorited_at_timesta
    fun getFavoriteChannels(): Flow<List<FavoriteChannelEntity>>

    @Query("SELECT id FROM favorite_channels WHERE is_deleted = 0")
    fun getFavoriteChannelIds(): Flow<List<String>>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insert(channel: FavoriteChannelEntity)

    /**
     * Marks a channel for deletion. It will be removed from the UI
     * and queued for deletion on the server during the next sync.
     * --- FIX: Update the query to use the PENDING_DELETE enum value ---
     */
    @Query("UPDATE favorite_channels SET is_deleted = 1, sync_status = 'PENDING_DELETE' WHERE
    suspend fun softDelete(channelId: String)

    @Query("SELECT EXISTS(SELECT 1 FROM favorite_channels WHERE id = :channelId AND is_deleted
```

```
    fun isFavorited(channelId: String): Flow<Boolean>

    // --- NEW METHODS FOR SYNC ---
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun upsert(channels: List<FavoriteChannelEntity>)

    // --- FIX: Update the query to use the DIRTY enum value ---
    @Query("SELECT * FROM favorite_channels WHERE sync_status = 'DIRTY'")
    suspend fun getDirtyItems(): List<FavoriteChannelEntity>

    // --- FIX: Update the query to use the PENDING_DELETE enum value ---
    @Query("SELECT id FROM favorite_channels WHERE sync_status = 'PENDING_DELETE'")
    suspend fun getPendingDeletionIds(): List<String>

    // --- FIX: Update the query to use the SYNCED enum value ---
    @Query("UPDATE favorite_channels SET sync_status = 'SYNCED' WHERE id IN (:channelIds)")
    suspend fun markAsSynced(channelIds: List<String>)

    @Query("DELETE FROM favorite_channels WHERE id IN (:channelIds) AND sync_status = 'PENDING
    suspend fun deleteSyncedDeletions(channelIds: List<String>)
}

// File: java\com\example\holodex\data\db\HistoryDao.kt
// File: java/com/example/holodex/data/db/HistoryDao.kt (MODIFIED)
package com.example.holodex.data.db

import androidx.room.Dao
import androidx.room.Insert
import androidx.room.OnConflictStrategy
import androidx.room.Query
import androidx.room.Transaction
import kotlinx.coroutines.flow.Flow

@Dao
interface HistoryDao {

    @Query("SELECT * FROM history_items ORDER BY playedAtTimestamp DESC")
    fun getHistory(): Flow<List<HistoryItemEntity>>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insert(item: HistoryItemEntity)

    @Query("DELETE FROM history_items WHERE itemId = :itemId")
    suspend fun deleteByItemId(itemId: String)

    @Transaction
    suspend fun upsert(item: HistoryItemEntity) {
        deleteByItemId(item.itemId)
        insert(item)
    }

    @Query("DELETE FROM history_items")
    suspend fun clearAll()
```

```
}

// File: java\com\example\holodex\data\db\HolodexSongListConverter.kt
// File: com/example/holodex/data/db/HolodexSongListConverter.kt
package com.example.holodex.data.db

import androidx.room.TypeConverter
import com.example.holodex.data.model.HolodexSong
import com.google.gson.Gson
import com.google.gson.reflect.TypeToken

class HolodexSongListConverter {
    private val gson = Gson()

    @TypeConverter
    fun fromHolodexSongList(songs: List<HolodexSong>?): String? {
        return songs?.let { gson.toJson(it) }
    }

    @TypeConverter
    fun toHolodexSongList(songsJson: String?): List<HolodexSong>? {
        return songsJson?.let {
            val listType = object : TypeToken<List<HolodexSong>>() {}.type
            gson.fromJson(it, listType)
        }
    }
}

// File: java\com\example\holodex\data\db\LikedItemDao.kt
// File: java/com/example/holodex/data/db/LikedItemDao.kt

package com.example.holodex.data.db

import androidx.room.Dao
import androidx.room.Insert
import androidx.room.OnConflictStrategy
import androidx.room.Query
import kotlinx.coroutines.flow.Flow

@Dao
interface LikedItemDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insert(likedItem: LikedItemEntity)

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insert(likedItems: List<LikedItemEntity>)

    @Query("SELECT * FROM liked_items WHERE sync_status != 'SYNCED'")
    suspend fun getUnsyncedItems(): List<LikedItemEntity>

    @Query("SELECT * FROM liked_items WHERE sync_status = 'DIRTY' AND server_id IS NULL")
    suspend fun getOrphanedDirtyItems(): List<LikedItemEntity>

    @Query("DELETE FROM liked_items WHERE itemId = :itemId")
    suspend fun deleteByItemId(itemId: String)
```

```
@Query("SELECT EXISTS(SELECT 1 FROM liked_items WHERE itemId = :itemId AND sync_status !=
fun isLiked(itemId: String): Flow<Boolean>

@Query("SELECT * FROM liked_items WHERE itemId = :itemId LIMIT 1")
suspend fun getLikedItem(itemId: String): LikedItemEntity?

@Query("SELECT * FROM liked_items WHERE sync_status != 'PENDING_DELETE' ORDER BY liked_at
fun getAllLikedItemsSortedByDate(): Flow<List<LikedItemEntity>>

@Query("SELECT * FROM liked_items")
suspend fun getAllLikedItemsOnce(): List<LikedItemEntity>

@Query("DELETE FROM liked_items WHERE sync_status = 'SYNCED' AND item_type = 'SONG_SEGMENT
suspend fun deleteAllSyncedSongSegments()

@Query("UPDATE liked_items SET sync_status = :status, last_modified_at = :timestamp WHERE
suspend fun updateStatusAndTimestamp(itemId: String, status: SyncStatus, timestamp: Long)

@Query("SELECT * FROM liked_items WHERE item_type = 'VIDEO' AND sync_status != 'PENDING_DE
fun getFavoritedVideosSortedByDate(): Flow<List<LikedItemEntity>>

@Query("SELECT * FROM liked_items WHERE item_type = 'SONG_SEGMENT' AND sync_status != 'PEN
fun getLikedSongSegmentsSortedByDate(): Flow<List<LikedItemEntity>>

@Query("SELECT DISTINCT videoId FROM liked_items WHERE sync_status != 'PENDING_DELETE'")
fun getAllDistinctLikedVideoIds(): Flow<List<String>>

@Query("DELETE FROM liked_items")
suspend fun clearAll()

@Query("UPDATE liked_items SET sync_status = 'PENDING_DELETE', last_modified_at = :timesta
suspend fun markForDeletion(itemId: String, timestamp: Long = System.currentTimeMillis())

@Query("SELECT * FROM liked_items WHERE item_type = :itemTypeName AND sync_status != 'PEND
fun getLikedItemsByTypeSortedByDate(itemTypeName: String): Flow<List<LikedItemEntity>>

@Query("SELECT * FROM liked_items WHERE item_type = :itemTypeName AND sync_status != 'PEND
fun getLikedItemsByTypePaged(itemTypeName: String, limit: Int, offset: Int): Flow<List<Lik

@Query("SELECT COUNT(itemId) FROM liked_items WHERE item_type = :itemTypeName AND sync_sta
suspend fun countLikedItemsByType(itemTypeName: String): Int

@Query("UPDATE liked_items SET sync_status = 'PENDING_DELETE', last_modified_at = :timesta
suspend fun performMarkForDeletion(itemId: String, timestamp: Long)

@Query("SELECT * FROM liked_items WHERE sync_status = 'DIRTY'")
suspend fun getDirtyItems(): List<LikedItemEntity>

@Query("SELECT itemId FROM liked_items WHERE sync_status = 'PENDING_DELETE'")
suspend fun getPendingDeletionIds(): List<String>

@Query("UPDATE liked_items SET sync_status = 'SYNCED' WHERE itemId IN (:itemIds)")
suspend fun markAsSynced(itemIds: List<String>)
```

```
    @Query("DELETE FROM liked_items WHERE itemId IN (:itemIds) AND sync_status = 'PENDING_DELE
    suspend fun deleteSyncedDeletions(itemIds: List<String>)

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun upsert(items: List<LikedItemEntity>)

    @Query("SELECT itemId FROM liked_items WHERE itemId IN (:itemIds)")
    suspend fun getLikedItemIds(itemIds: List<String>): List<String>
}

// File: java\com\example\holodex\data\db\LocalDao.kt
// File: java/com/example/holodex/data/db/LocalDao.kt

package com.example.holodex.data.db

import androidx.room.Dao
import androidx.room.Insert
import androidx.room.OnConflictStrategy
import androidx.room.Query
import kotlinx.coroutines.flow.Flow

@Dao
interface LocalDao {

    // --- Local Favorites ---
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun addLocalFavorite(favorite: LocalFavoriteEntity)

    @Query("DELETE FROM local_favorites WHERE itemId = :itemId")
    suspend fun removeLocalFavorite(itemId: String)

    @Query("SELECT * FROM local_favorites")
    fun getLocalFavorites(): Flow<List<LocalFavoriteEntity>>

    @Query("SELECT itemId FROM local_favorites")
    fun getLocalFavoriteIds(): Flow<List<String>>

    // --- External Channels ---
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun addExternalChannel(channel: ExternalChannelEntity)

    @Query("DELETE FROM external_channels WHERE channelId = :channelId")
    suspend fun removeExternalChannel(channelId: String)

    @Query("SELECT * FROM external_channels")
    fun getAllExternalChannels(): Flow<List<ExternalChannelEntity>>

    @Query("SELECT * FROM external_channels WHERE channelId = :channelId")
    suspend fun getExternalChannel(channelId: String): ExternalChannelEntity?

    // --- Local Playlists (NEW) ---

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun createLocalPlaylist(playlist: LocalPlaylistEntity): Long
```

```
    @Query("SELECT * FROM local_playlists")
    fun getAllLocalPlaylists(): Flow<List<LocalPlaylistEntity>>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun addSongToLocalPlaylist(item: LocalPlaylistItemEntity)

    @Query("SELECT * FROM local_playlist_items WHERE playlistOwnerId = :playlistId ORDER BY it
    fun getItemsForLocalPlaylist(playlistId: Long): Flow<List<LocalPlaylistItemEntity>>
}

// File: java\com\example\holodex\data\db\LocalEntities.kt
// File: java/com/example/holodex/data/db/LocalEntities.kt

package com.example.holodex.data.db

import androidx.room.ColumnInfo

import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "local_favorites")
data class LocalFavoriteEntity(
    @PrimaryKey val itemId: String,
    val videoId: String,
    val channelId: String,
    val title: String,
    val artistText: String,
    val artworkUrl: String?,
    val durationSec: Long,
    val isSegment: Boolean,
    val songStartSec: Int?,
    val songEndSec: Int?
)

@Entity(tableName = "external_channels")
data class ExternalChannelEntity(
    @PrimaryKey val channelId: String,
    val name: String,
    val photoUrl: String?,
    val lastCheckedTimestamp: Long = 0,
    val status: String = "OK",
    val errorCount: Int = 0
)

@Entity(tableName = "local_playlists")
data class LocalPlaylistEntity(
    @PrimaryKey(autoGenerate = true) val localPlaylistId: Long = 0,
    val name: String,
    val description: String?,
    val createdAt: Long = System.currentTimeMillis()
)

@Entity(
    tableName = "local_playlist_items",
    primaryKeys = ["playlistOwnerId", "itemId"]
```

```
)
data class LocalPlaylistItemEntity(
    val playlistOwnerId: Long,
    val itemId: String, // The composite ID: "videoId_startTime"
    val videoId: String,
    val itemOrder: Int,

    // Snapshot data for quick display
    val title: String,
    val artistText: String,
    val artworkUrl: String?,
    val durationSec: Long,
    val channelId: String,
    val isSegment: Boolean,
    val songStartSec: Int?,
    val songEndSec: Int?
)


// File: java\com\example\holodex\data\db\ParentVideoMetadataDao.kt
// File: java/com/example/holodex/data/db/ParentVideoMetadataDao.kt
package com.example.holodex.data.db

import androidx.room.Dao
import androidx.room.Insert
import androidx.room.OnConflictStrategy
import androidx.room.Query

@Dao
interface ParentVideoMetadataDao {
    @Insert(onConflict = OnConflictStrategy.IGNORE)
    suspend fun insert(metadata: ParentVideoMetadataEntity)

    @Query("SELECT * FROM parent_video_metadata WHERE videoId = :videoId")
    suspend fun getById(videoId: String): ParentVideoMetadataEntity?
}

// File: java\com\example\holodex\data\db\PlaylistDao.kt
// File: java/com/example/holodex/data/db/PlaylistDao.kt

package com.example.holodex.data.db

import androidx.room.Dao
import androidx.room.Insert
import androidx.room.OnConflictStrategy
import androidx.room.Query
import androidx.room.Transaction
import androidx.room.Update
import kotlinx.coroutines.flow.Flow

@Dao
interface PlaylistDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertPlaylist(playlist: PlaylistEntity): Long
```

```
// --- START OF FIX: Add a dedicated @Update function ---
@Update
suspend fun updatePlaylist(playlist: PlaylistEntity)
// --- END OF FIX ---

@Transaction
suspend fun updatePlaylistAndItems(playlist: PlaylistEntity, items: List<PlaylistItemEntit
    updatePlaylist(playlist)
    deleteAllItemsForPlaylist(playlist.playlistId)
    if (items.isNotEmpty()) {
        upsertPlaylistItems(items)
    }
}

@Query("SELECT * FROM playlists WHERE is_deleted = 0 ORDER BY name ASC")
fun getAllPlaylists(): Flow<List<PlaylistEntity>>

@Query("SELECT * FROM playlists WHERE playlistId = :playlistId")
suspend fun getPlaylistById(playlistId: Long): PlaylistEntity?

@Query("UPDATE playlists SET is_deleted = 1, sync_status = 'PENDING_DELETE' WHERE playlist
suspend fun softDeletePlaylist(playlistId: Long)

@Query("DELETE FROM playlists WHERE playlistId = :playlistId")
suspend fun deletePlaylist(playlistId: Long)

// --- START OF FIX: Change strategy to REPLACE for robustness ---
@Insert(onConflict = OnConflictStrategy.REPLACE)
suspend fun insertPlaylistItem(playlistItem: PlaylistItemEntity)
// --- END OF FIX ---

@Query("UPDATE playlist_items SET sync_status = 'PENDING_DELETE' WHERE playlist_owner_id =
suspend fun softDeletePlaylistItem(playlistId: Long, itemIdInPlaylist: String)

@Query("SELECT * FROM playlist_items WHERE playlist_owner_id = :playlistId AND sync_status
fun getItemsForPlaylist(playlistId: Long): Flow<List<PlaylistItemEntity>>

// --- START OF FIX: Make query more robust by excluding soft-deleted items ---
@Query("SELECT MAX(item_order) FROM playlist_items WHERE playlist_owner_id = :playlistId A
suspend fun getLastItemOrder(playlistId: Long): Int?
// --- END OF FIX ---

@Query("SELECT * FROM playlists")
suspend fun getAllPlaylistsOnce(): List<PlaylistEntity>

@Query("SELECT * FROM playlists WHERE sync_status != 'SYNCED'")
suspend fun getUnsyncedPlaylists(): List<PlaylistEntity>

@Query("SELECT * FROM playlist_items WHERE sync_status != 'SYNCED'")
suspend fun getUnsyncedPlaylistItems(): List<PlaylistItemEntity>

@Insert(onConflict = OnConflictStrategy.REPLACE)
suspend fun upsertPlaylists(playlists: List<PlaylistEntity>)

@Insert(onConflict = OnConflictStrategy.REPLACE)
```

```
    suspend fun upsertPlaylistItems(items: List<PlaylistItemEntity>)

    @Query("DELETE FROM playlists WHERE is_deleted = 1")
    suspend fun deleteSoftDeletedPlaylists()

    @Query("DELETE FROM playlist_items WHERE playlist_owner_id = :playlistId")
    suspend fun deleteAllItemsForPlaylist(playlistId: Long)

    @Query("UPDATE playlist_items SET sync_status = 'SYNCED' WHERE playlist_owner_id = :playli
    suspend fun markItemsAsSynced(playlistId: Long, itemIds: List<String>)

    @Query("DELETE FROM playlist_items WHERE sync_status = 'PENDING_DELETE' AND playlist_owner
    suspend fun deleteSyncedSoftDeletedItemsForPlaylist(playlistId: Long)

    @Query("UPDATE playlists SET name = :name, description = :description, updated_at = :times
    suspend fun updatePlaylistMetadata(playlistId: Long, name: String?, description: String?,

}

// File: java\com\example\holodex\data\db\SearchPageDao.kt
package com.example.holodex.data.db

import androidx.room.Dao
import androidx.room.Insert
import androidx.room.OnConflictStrategy
import androidx.room.Query

@Dao
interface SearchPageDao {
    @Query("SELECT * FROM cached_search_pages WHERE pageKey = :pageKey")
    suspend fun getPage(pageKey: String): CachedSearchPage?

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertPage(page: CachedSearchPage)

    @Query("DELETE FROM cached_search_pages WHERE pageKey = :pageKey")
    suspend fun deletePage(pageKey: String)

    /**
     * Deletes pages older than the given timestamp for search caches.
     */
    @Query("DELETE FROM cached_search_pages WHERE timestamp < :expiredTime")
    suspend fun deleteExpiredSearchPages(expiredTime: Long)

    /**
     * Deletes all pages from the search cache.
     */
    @Query("DELETE FROM cached_search_pages")
    suspend fun deleteAllSearchPages()

    @Query("SELECT COUNT(pageKey) FROM cached_search_pages")
    suspend fun getSearchCacheSize(): Int
}

// File: java\com\example\holodex\data\db\StarredPlaylistDao.kt
```

```
// File: java/com/example/holodex/data/db/StarredPlaylistDao.kt (NEW FILE)
package com.example.holodex.data.db

import androidx.room.Dao
import androidx.room.Insert
import androidx.room.OnConflictStrategy
import androidx.room.Query
import kotlinx.coroutines.flow.Flow

@Dao
interface StarredPlaylistDao {
    @Query("SELECT * FROM starred_playlists")
    fun getStarredPlaylists(): Flow<List<StarredPlaylistEntity>>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insert(starredPlaylist: StarredPlaylistEntity)

    @Query("DELETE FROM starred_playlists WHERE playlist_id = :playlistId")
    suspend fun deleteById(playlistId: String)

    // --- Methods for Sync Logic ---
    @Query("SELECT * FROM starred_playlists WHERE sync_status != 'SYNCED'")
    suspend fun getUnsyncedItems(): List<StarredPlaylistEntity>

    @Query("DELETE FROM starred_playlists WHERE sync_status = 'SYNCED'")
    suspend fun deleteAllSyncedItems()

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun upsertAll(items: List<StarredPlaylistEntity>)
}

// File: java\com\example\holodex\data\db\StarredPlaylistEntity.kt
// File: java/com/example/holodex/data/db/StarredPlaylistEntity.kt (NEW FILE)
package com.example.holodex.data.db

import androidx.room.ColumnInfo
import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "starred_playlists")
data class StarredPlaylistEntity(
    @PrimaryKey
    @ColumnInfo(name = "playlist_id")
    val playlistId: String,

    @ColumnInfo(name = "sync_status")
    val syncStatus: SyncStatus
)

// File: java\com\example\holodex\data\db\SyncMetadataDao.kt
// File: java/com/example/holodex/data/db/SyncMetadataDao.kt
// (Create this new file)

package com.example.holodex.data.db
```

```
import androidx.room.Dao
import androidx.room.Insert
import androidx.room.OnConflictStrategy
import androidx.room.Query

@Dao
interface SyncMetadataDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun setLastSyncTimestamp(metadata: SyncMetadataEntity)

    @Query("SELECT lastSyncTimestamp FROM sync_metadata WHERE dataType = :dataType")
    suspend fun getLastSyncTimestamp(dataType: String): Long?
}

// File: java\com\example\holodex\data\db\VideoDao.kt
package com.example.holodex.data.db

import androidx.room.Dao
import androidx.room.Insert
import androidx.room.OnConflictStrategy
import androidx.room.Query
import androidx.room.Transaction
import kotlinx.coroutines.flow.Flow

@Dao
interface VideoDao {

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertVideo(video: CachedVideoEntity)

    @Transaction
    @Query("SELECT * FROM videos WHERE id = :videoId")
    suspend fun getVideoWithSongsOnce(videoId: String): VideoWithSongs?

    @Query("SELECT * FROM videos WHERE id = :videoId LIMIT 1")
    suspend fun getVideoByIdOnce(videoId: String): CachedVideoEntity?


    @Query("DELETE FROM videos")
    fun clearAllVideos()

    @Query("DELETE FROM songs WHERE video_id = :videoId")
    suspend fun deleteSongsForVideo(videoId: String)

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertSongs(songs: List<CachedSongEntity>)

    @Query("SELECT * FROM videos WHERE id IN (:videoIds)")
    fun getVideosByIds(videoIds: List<String>): Flow<List<CachedVideoEntity>>


    @Query("DELETE FROM songs")
    fun clearAllSongs()
}
```

```
// File: java\com\example\holodex\data\db\mappers\SyncMappers.kt
// File: java\com\example\holodex\data\db\mappers\SyncMappers.kt (NEW FILE)
package com.example.holodex.data.db.mappers

import com.example.holodex.data.api.FavoriteChannelApiDto
import com.example.holodex.data.api.LikedSongApiDto
import com.example.holodex.data.api.PlaylistDto
import com.example.holodex.data.db.FavoriteChannelEntity
import com.example.holodex.data.db.HistoryItemEntity
import com.example.holodex.data.db.LikedItemEntity
import com.example.holodex.data.db.LikedItemType
import com.example.holodex.data.db.PlaylistEntity
import com.example.holodex.data.db.SyncStatus
import com.example.holodex.data.model.HolodexVideoItem
import com.example.holodex.data.model.discovery.MusicdexSong
import timber.log.Timber

fun LikedSongApiDto.toLikedItemEntity(parentVideo: HolodexVideoItem): LikedItemEntity {
    return LikedItemEntity(
        itemId = this.id,
        serverId = this.id,
        videoId = this.video_id,
        itemType = LikedItemType.SONG_SEGMENT,
        titleSnapshot = this.name,
        artistTextSnapshot = parentVideo.channel.name,
        albumTextSnapshot = parentVideo.title,
        artworkUrlSnapshot = this.channel?.photo ?: parentVideo.channel.photoUrl,
        descriptionSnapshot = parentVideo.description,
        channelIdSnapshot = this.channel_id,
        durationSecSnapshot = (this.end - this.start).toLong(),
        actualSongName = this.name,
        actualSongArtist = this.original_artist,
        actualSongArtworkUrl = null, // This specific art isn't in the liked response
        songStartSeconds = this.start,
        songEndSeconds = this.end,
        syncStatus = SyncStatus.SYNCED,
        lastModifiedAt = System.currentTimeMillis()
    )
}

fun FavoriteChannelApiDto.toFavoriteChannelEntity(): FavoriteChannelEntity {
    return FavoriteChannelEntity(
        id = this.id,
        name = this.name ?: this.english_name
        ?: "Unknown Channel",
        englishName = this.english_name,
        photoUrl = this.photo,
        org = this.org,
        subscriberCount = null,
        twitter = this.twitter,
        syncStatus = SyncStatus.SYNCED,
        isDeleted = false,
        favoritedAtTimestamp = System.currentTimeMillis()
```

```
    )
}


fun LikedSongApiDto.toLikedItemEntityShell(): LikedItemEntity {
    return LikedItemEntity(
        itemId = LikedItemEntity.generateSongItemId(this.video_id, this.start), // Local compo
        serverId = this.id, // <-- SAVE THE SERVER UUID
        videoId = this.video_id,
        itemType = LikedItemType.SONG_SEGMENT,
        titleSnapshot = this.name,
        artistTextSnapshot = this.channel?.name ?: "Unknown Channel",
        albumTextSnapshot = "...", // Placeholder to be enriched
        artworkUrlSnapshot = this.art,
        descriptionSnapshot = null, // To be enriched
        channelIdSnapshot = this.channel_id,
        durationSecSnapshot = (this.end - this.start).toLong(),
        actualSongName = this.name,
        actualSongArtist = this.original_artist,
        actualSongArtworkUrl = this.art,
        songStartSeconds = this.start,
        songEndSeconds = this.end,
        syncStatus = SyncStatus.SYNCED, // Data comes from server, so it's synced
        lastModifiedAt = System.currentTimeMillis()
    )
}
fun MusicdexSong.toHistoryItemEntity(
    parentVideo: HolodexVideoItem,
    // --- START OF FIX: Add the new parameter ---
    syntheticPlayedAtTimestamp: Long
    // --- END OF FIX ---
): HistoryItemEntity? {
    // We no longer need to parse the 'ts' field as it doesn't exist.
    // We will use the provided synthetic timestamp directly.
    if (this.channelId.isNullOrBlank()) {
        Timber.w("Skipping history item ('${this.name}') because its top-level 'channel_id' is
        return null
    }

    return HistoryItemEntity(
        playedAtTimestamp = syntheticPlayedAtTimestamp, // <-- Use the passed-in value
        itemId = "${this.videoId}_${this.start}",
        videoId = this.videoId,
        songStartSeconds = this.start,
        title = this.name,
        artistText = this.channel.name,
        artworkUrl = this.artUrl,
        durationSec = (this.end - this.start).toLong(),
        channelId = this.channelId
    )
}
fun PlaylistDto.toEntity(): PlaylistEntity {
    return PlaylistEntity(
        playlistId = 0, // Let Room auto-generate the local ID
        serverId = this.id,
```

```
        name = this.title,
        description = this.description,
        owner = this.owner,
        type = this.type ?: "ugp",
        createdAt = this.createdAt,
        last_modified_at = this.updatedAt,
        isDeleted = false,
        syncStatus = SyncStatus.SYNCED // Data from server is considered synced
    )
}

// File: java\com\example\holodex\data\download\DownloadCompletionObserver.kt
package com.example.holodex.data.download

import android.content.Context
import androidx.annotation.OptIn
import androidx.media3.common.util.UnstableApi
import androidx.media3.exoplayer.offline.Download
import androidx.media3.exoplayer.offline.DownloadManager
import androidx.work.Data
import androidx.work.ExistingWorkPolicy
import androidx.work.OneTimeWorkRequestBuilder
import androidx.work.WorkManager
import com.example.holodex.background.M4AExportWorker
import com.example.holodex.data.db.DownloadStatus
import com.example.holodex.data.db.DownloadedItemDao
import com.example.holodex.data.db.ParentVideoMetadataDao
import dagger.hilt.android.qualifiers.ApplicationContext
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.SupervisorJob
import kotlinx.coroutines.launch
import timber.log.Timber
import javax.inject.Inject
import javax.inject.Singleton

@OptIn(UnstableApi::class)
@Singleton
class DownloadCompletionObserver @Inject constructor(
    @ApplicationContext private val context: Context,
    private val downloadManager: DownloadManager,
    private val downloadedItemDao: DownloadedItemDao,
    private val parentVideoMetadataDao: ParentVideoMetadataDao
) : DownloadManager.Listener {

    companion object {
        private const val TAG = "DownloadCompletionObs"
    }
    private val scope = CoroutineScope(Dispatchers.IO + SupervisorJob())
    private val workManager = WorkManager.getInstance(context)

    fun initialize() {
        downloadManager.addListener(this)
    }
```

```
override fun onDownloadChanged(
    manager: DownloadManager,
    download: Download,
    finalException: Exception?
) {
    scope.launch {
        val itemId = download.request.id

        // --- FIX: Perform null check at the top and return early ---
        val currentDbEntry = downloadedItemDao.getById(itemId)
        if (currentDbEntry == null) {
            Timber.w("$TAG: onDownloadChanged for ID: $itemId, but no corresponding DB ent
            return@launch
        }
        // --- END OF FIX ---

        // From this point on, the compiler knows `currentDbEntry` is not null.

        when (download.state) {
            Download.STATE_COMPLETED -> {
                Timber.i("$TAG: Media3 download COMPLETED for ID: $itemId. Target format:
                downloadedItemDao.updateStatus(itemId, DownloadStatus.PROCESSING)

                val parentVideoId = itemId.split('_').firstOrNull()
                val parentMetadata = parentVideoId?.let { parentVideoMetadataDao.getById(i

                if (parentMetadata == null) {
                    Timber.e("$TAG: Parent metadata not found for $itemId. Cannot proceed
                    downloadedItemDao.updateStatus(itemId, DownloadStatus.FAILED)
                    return@launch
                }

                val albumName = parentMetadata.title
                val startTimeSeconds = currentDbEntry.videoId.split('_').getOrNull(1)?.toL
                val clipStartMs = startTimeSeconds * 1000
                val clipEndMs = (startTimeSeconds + currentDbEntry.durationSec) * 1000
                val trackNumber = currentDbEntry.trackNumber

                when (currentDbEntry.targetFormat) {
                    "M4A", "" -> {
                        val workData = Data.Builder()
                            .putString(M4AExportWorker.KEY_ITEM_ID, itemId)
                            .putString(M4AExportWorker.KEY_ORIGINAL_URI, download.request.
                            .putString(M4AExportWorker.KEY_SONG_TITLE, currentDbEntry.titl
                            .putString(M4AExportWorker.KEY_ARTIST_NAME, currentDbEntry.art
                            .putString(M4AExportWorker.KEY_ALBUM_NAME, albumName)
                            .putString(M4AExportWorker.KEY_ARTWORK_URI, currentDbEntry.art
                            .putLong(M4AExportWorker.KEY_CLIP_START_MS, clipStartMs)
                            .putLong(M4AExportWorker.KEY_CLIP_END_MS, clipEndMs)
                            .apply { trackNumber?.let { putInt(M4AExportWorker.KEY_TRACK_N
                            .build()

                        val exportRequest = OneTimeWorkRequestBuilder<M4AExportWorker>()
                            .setInputData(workData)
                            .build()
```

```
                            workManager.enqueueUniqueWork("export_$itemId", ExistingWorkPolicy
                            Timber.d("$TAG: Enqueued M4AExportWorker for $itemId.")
                        }
                        else -> {
                            Timber.e("$TAG: Unknown target format '${currentDbEntry.targetForm
                            downloadedItemDao.updateStatus(itemId, DownloadStatus.FAILED)
                        }
                    }
                }
                Download.STATE_FAILED -> {
                    Timber.e(finalException, "$TAG: Media3 download FAILED for ID: $itemId. Re
                    downloadedItemDao.updateStatus(itemId, DownloadStatus.FAILED)
                }
                Download.STATE_STOPPED -> {
                    Timber.w("$TAG: Download STOPPED for ID: $itemId. Reason: ${download.stopR
                    // The C.LENGTH_UNSET is a Long, download.stopReason is Int. These should
                    // The original comparison was `if (download.stopReason != Download.STOP_R
                    // Download.STOP_REASON_NONE is an Int (0). This comparison is type-correc
                    if (download.stopReason != Download.STOP_REASON_NONE || finalException !=
                        downloadedItemDao.updateStatus(itemId, DownloadStatus.FAILED)
                    } else {
                        if(currentDbEntry?.downloadStatus == DownloadStatus.DOWNLOADING) {
                            // No specific action, let reconciliation handle if it doesn't res
                        }
                        Timber.d("$TAG: Download $itemId STOPPED without error, current DB sta
                    }
                }
                Download.STATE_REMOVING -> {
                    Timber.d("$TAG: Media3 download is being removed (ID: ${download.request.i
                }
                else -> {
                    Timber.d("$TAG: Download state ${download.state} (may be RESTARTING) for I
                    if (download.state == Download.STATE_RESTARTING && currentDbEntry?.downloa
                        downloadedItemDao.updateStatus(itemId, DownloadStatus.DOWNLOADING)
                    }
                }
            }
        }
    }
}

// File: java\com\example\holodex\data\download\DownloadExceptions.kt
// File: java/com/example/holodex/data/download/DownloadExceptions.kt
package com.example.holodex.data.download

/**
 * A specific exception thrown when a download is initiated without a
 * download location being configured in the app's settings.
 */
class NoDownloadLocationException(message: String) : Exception(message)

// File: java\com\example\holodex\data\download\LegacyDownloadScanner.kt
package com.example.holodex.data.download
```

```
import android.content.Context
import android.media.MediaMetadataRetriever
import android.net.Uri
import android.provider.MediaStore
import com.example.holodex.data.db.DownloadStatus
import com.example.holodex.data.db.DownloadedItemDao
import com.example.holodex.data.db.DownloadedItemEntity
import com.example.holodex.data.model.HolodexSong
import com.example.holodex.data.model.HolodexVideoItem
import com.example.holodex.data.model.SearchCondition
import com.example.holodex.data.model.VideoSearchRequest
import com.example.holodex.data.repository.HolodexRepository
import dagger.hilt.android.qualifiers.ApplicationContext
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.async
import kotlinx.coroutines.awaitAll
import kotlinx.coroutines.coroutineScope
import kotlinx.coroutines.flow.first
import kotlinx.coroutines.sync.Semaphore
import kotlinx.coroutines.withContext
import timber.log.Timber
import java.io.File
import javax.inject.Inject
import javax.inject.Singleton
import kotlin.math.max

@Singleton
class LegacyDownloadScanner @Inject constructor(
    @ApplicationContext private val context: Context,
    private val holodexRepository: HolodexRepository,
    private val downloadedItemDao: DownloadedItemDao
) {
    companion object {
        private const val TAG = "LegacyDownloadScanner"
        private const val DOWNLOAD_FOLDER_NAME = "HolodexMusic"
        private const val SIMILARITY_THRESHOLD = 0.85 // 85% similarity needed for a match
        private const val CONCURRENT_SCANS_LIMIT = 4 // Process up to 4 videos at a time
    }

    private data class FileInfo(val uri: Uri, val name: String, val lastModified: Long)
    private data class FileMetadata(val title: String, val artist: String, val album: String)
    private data class FileWithMetadata(val fileInfo: FileInfo, val metadata: FileMetadata)

    suspend fun scanAndImportLegacyDownloads(): Int = withContext(Dispatchers.IO) {
        var totalImportedCount = 0
        try {
            val potentialLegacyFiles = queryMediaStoreForAppDownloads()
            if (potentialLegacyFiles.isEmpty()) {
                Timber.d("$TAG: No .m4a files found via MediaStore.")
                return@withContext 0
            }

            val existingDbFiles = downloadedItemDao.getAllDownloads().first().map { it.fileNam
            val filesToProcess = potentialLegacyFiles.filterNot { existingDbFiles.contains(it.
```

```
        if (filesToProcess.isEmpty()) {
            Timber.i("$TAG: All ${potentialLegacyFiles.size} files are already in the data
            return@withContext 0
        }

        Timber.i("$TAG: Found ${filesToProcess.size} new potential legacy files. Grouping

        val filesGroupedByAlbum = filesToProcess.mapNotNull { fileInfo ->
            extractMetadata(fileInfo.uri)?.let { metadata -> FileWithMetadata(fileInfo, me
        }.groupBy { it.metadata.album }

        Timber.d("$TAG: Grouped files into ${filesGroupedByAlbum.size} potential videos. S

        val semaphore = Semaphore(CONCURRENT_SCANS_LIMIT)

        totalImportedCount = coroutineScope {
            val deferredImports = filesGroupedByAlbum.map { (albumTitle, filesInAlbum) ->
                async {
                    var groupImportCount = 0
                    semaphore.acquire() // Wait for a permit to start
                    try {
                        val artist = filesInAlbum.first().metadata.artist
                        val videoWithSongs = findVideoForGroup(albumTitle, artist)

                        if (videoWithSongs != null) {
                            for (fileWithMeta in filesInAlbum) {
                                val matchedSong = findMatchingSong(fileWithMeta.metadata.t
                                if (matchedSong != null) {
                                    importSong(fileWithMeta.fileInfo, videoWithSongs, matc
                                    groupImportCount++
                                } else {
                                    Timber.w("$TAG: FAILED to find song match for '${fileW
                                }
                            }
                            if (groupImportCount > 0) {
                                Timber.i("$TAG: Imported $groupImportCount songs for video
                            }
                        } else {
                            Timber.w("$TAG: FAILED to find a parent video for group with a
                        }
                    } finally {
                        semaphore.release() // Always release the permit
                    }
                    groupImportCount // Return the count for this group
                }
            }
            deferredImports.awaitAll().sum() // Wait for all groups to finish and sum the
        }

    } catch (e: Exception) {
        Timber.e(e, "$TAG: An error occurred during the scan process.")
    }
    Timber.i("$TAG: Scan complete. Imported a total of $totalImportedCount new files.")
    return@withContext totalImportedCount
}
```

```
private suspend fun findVideoForGroup(albumTitle: String, artist: String): HolodexVideoIte
    val organization = extractOrgFromArtist(artist)
    val coreTitle = extractCoreTitle(albumTitle)

    // Search for both the unique part of the title AND the artist name
    val searchRequest = VideoSearchRequest(
        target = listOf("stream", "clip"),
        conditions = listOf(
            SearchCondition(text = coreTitle),
            SearchCondition(text = artist)
        ),
        org = organization?.let { listOf(it) }
    )

    try {
        val searchResult = holodexRepository.holodexApiService.searchVideosAdvanced(search

        val potentialVideos = searchResult.body()?.items
        if (potentialVideos.isNullOrEmpty()) {
            return null // API found no candidates
        }

        // Client-side filter: find the video in the results that is most similar to our f
        val bestVideoMatch = potentialVideos.maxByOrNull {
            calculateSimilarity(it.title, albumTitle)
        } ?: return null

        // Confidence check: If the best match is still not very similar, reject it.
        if (calculateSimilarity(bestVideoMatch.title, albumTitle) < 0.6) {
            Timber.w("$TAG: Found a potential video ('${bestVideoMatch.title}'), but it wa
            return null
        }

        val videoWithSongsResult = holodexRepository.getVideoWithSongs(bestVideoMatch.id)
        return videoWithSongsResult.getOrNull()
    } catch (e: Exception) {
        Timber.e(e, "$TAG: API error while finding video for group: $albumTitle")
        return null
    }
}

private fun findMatchingSong(fileTitle: String, apiSongs: List<HolodexSong>?): HolodexSong
    if (apiSongs.isNullOrEmpty()) return null

    val normalizedFileTitle = normalize(fileTitle)
    return apiSongs
        .map { apiSong ->
            val normalizedApiTitle = normalize(apiSong.name)
            val similarity = calculateSimilarity(normalizedFileTitle, normalizedApiTitle)
            apiSong to similarity
        }
        .filter { it.second >= SIMILARITY_THRESHOLD }
        .maxByOrNull { it.second }
        ?.first
```

```
}

private suspend fun importSong(
    fileInfo: FileInfo,
    video: HolodexVideoItem,
    song: HolodexSong
) {
    val entity = DownloadedItemEntity(
        videoId = "${video.id}_${song.start}",
        title = song.name,
        artistText = video.channel.name,
        channelId = video.channel.id ?: "unknown",
        artworkUrl = song.artUrl ?: video.channel.photoUrl,
        durationSec = (song.end - song.start).toLong(),
        localFileUri = fileInfo.uri.toString(),
        downloadStatus = DownloadStatus.COMPLETED,
        downloadedAt = fileInfo.lastModified,
        fileName = fileInfo.name,
        targetFormat = "M4A",
        downloadId = null,
        progress = 100,
        trackNumber = null
    )
    downloadedItemDao.insertOrUpdate(entity)
}

private fun queryMediaStoreForAppDownloads(): List<FileInfo> {
    val files = mutableListOf<FileInfo>()
    val projection = arrayOf(
        MediaStore.Audio.Media._ID,
        MediaStore.Audio.Media.DISPLAY_NAME,
        MediaStore.Audio.Media.DATE_MODIFIED
    )

    val selection = "${MediaStore.Audio.Media.RELATIVE_PATH} LIKE ?"
    val selectionArgs = arrayOf("%${File.separator}$DOWNLOAD_FOLDER_NAME${File.separator}%

    val sortOrder = "${MediaStore.Audio.Media.DATE_MODIFIED} DESC"
    val queryUri = MediaStore.Audio.Media.EXTERNAL_CONTENT_URI

    try {
        context.contentResolver.query(
            queryUri,
            projection,
            selection,
            selectionArgs,
            sortOrder
        )?.use { cursor ->
            val idColumn = cursor.getColumnIndexOrThrow(MediaStore.Audio.Media._ID)
            val nameColumn = cursor.getColumnIndexOrThrow(MediaStore.Audio.Media.DISPLAY_N
            val dateModifiedColumn = cursor.getColumnIndexOrThrow(MediaStore.Audio.Media.D

            while (cursor.moveToNext()) {
                val id = cursor.getLong(idColumn)
                val name = cursor.getString(nameColumn)
```

```
                val dateModified = cursor.getLong(dateModifiedColumn)
                val contentUri = Uri.withAppendedPath(queryUri, id.toString())
                files.add(FileInfo(contentUri, name, dateModified * 1000))
            }
        }
    } catch (e: Exception) {
        Timber.e(e, "$TAG: Failed to query MediaStore.")
    }
    return files
}

private fun extractMetadata(uri: Uri): FileMetadata? {
    return try {
        MediaMetadataRetriever().use { retriever ->
            retriever.setDataSource(context, uri)
            val title = retriever.extractMetadata(MediaMetadataRetriever.METADATA_KEY_TITL
            val artist = retriever.extractMetadata(MediaMetadataRetriever.METADATA_KEY_ART
            val album = retriever.extractMetadata(MediaMetadataRetriever.METADATA_KEY_ALBU

            if (title.isNullOrBlank() || artist.isNullOrBlank() || album.isNullOrBlank())
                Timber.w("$TAG: File at uri '$uri' is missing essential metadata.")
                null
            } else {
                FileMetadata(title, artist, album)
            }
        }
    } catch (e: Exception) {
        Timber.e(e, "$TAG: Failed to extract metadata from uri '$uri'")
        null
    }
}

private fun extractOrgFromArtist(artist: String): String? {
    return when {
        artist.contains("?????", ignoreCase = true) -> "Nijisanji"
        artist.contains("hololive", ignoreCase = true) -> "Hololive"
        else -> null
    }
}

private fun normalize(input: String): String {
    return input
        .lowercase()
        .replace(Regex("[\\s.,?/()!\\[\\]_{}\"']"), "")
}

private fun calculateSimilarity(s1: String, s2: String): Double {
    val jaro = jaroDistance(s1, s2)
    if (jaro < 0.7) return jaro
    var prefix = 0
    for (i in 0 until minOf(s1.length, s2.length, 4)) {
        if (s1[i] == s2[i]) prefix++ else break
    }
    return jaro + 0.1 * prefix * (1.0 - jaro)
}
```

```
private fun jaroDistance(s1: String, s2: String): Double {
    if (s1 == s2) return 1.0
    val len1 = s1.length
    val len2 = s2.length
    if (len1 == 0 || len2 == 0) return 0.0
    val matchDistance = max(len1, len2) / 2 - 1
    val s1Matches = BooleanArray(len1)
    val s2Matches = BooleanArray(len2)
    var matches = 0
    for (i in 0 until len1) {
        val start = max(0, i - matchDistance)
        val end = minOf(i + matchDistance + 1, len2)
        for (j in start until end) {
            if (!s2Matches[j] && s1[i] == s2[j]) {
                s1Matches[i] = true
                s2Matches[j] = true
                matches++
                break
            }
        }
    }
    if (matches == 0) return 0.0
    var t = 0.0
    var k = 0
    for (i in 0 until len1) {
        if (s1Matches[i]) {
            while (!s2Matches[k]) k++
            if (s1[i] != s2[k]) t++
            k++
        }
    }
    val transpositions = t / 2
    return (matches.toDouble() / len1 + matches.toDouble() / len2 + (matches - transpositi
}

private fun extractCoreTitle(albumTitle: String): String {
    // Priority 1: Extract content from Japanese brackets ??
    val bracketRegex = Regex("?(.*?)?")
    val bracketMatch = bracketRegex.find(albumTitle)
    if (bracketMatch != null && bracketMatch.groupValues[1].isNotBlank()) {
        return bracketMatch.groupValues[1]
            .replace("#", "")
            .trim()
    }

    // Priority 2: Split by Japanese punctuation to find meaningful phrases
    val punctuationRegex = Regex("[??]")
    val phrases = albumTitle.split(punctuationRegex)
        .map { it.trim() }
        .filter { it.length > 2 }

    if (phrases.isNotEmpty()) {
        return phrases.maxByOrNull { it.length } ?: albumTitle
    }
```

```
        // Fallback: Return full title if no punctuation found
        return albumTitle
    }
} // <-- This closing brace was missing.

// File: java\com\example\holodex\data\model\AudioStreamDetails.kt
package com.example.holodex.data.model // Make sure this package name matches your structure

import com.google.gson.annotations.SerializedName

// Represents what Musicdex or a similar service might return
data class AudioStreamDetails(
    @SerializedName("url") val streamUrl: String, // Direct audio stream URL
    @SerializedName("format") val format: String?, // e.g., "m4a", "opus"
    @SerializedName("quality") val quality: String? // e.g., "128kbps"
)

// File: java\com\example\holodex\data\model\ChannelSearchResult.kt
// File: java/com/example/holodex/data/model/ChannelSearchResult.kt
package com.example.holodex.data.model

data class ChannelSearchResult(
    val channelId: String,
    val name: String,
    val thumbnailUrl: String?,
    val subscriberCount: String?
)

// File: java\com\example\holodex\data\model\HolodexSong.kt
package com.example.holodex.data.model

import android.os.Parcelable
import com.google.gson.annotations.SerializedName
import kotlinx.parcelize.Parcelize

@Parcelize
data class HolodexSong(
    @SerializedName("name") val name: String,
    @SerializedName("start") val start: Int,
    @SerializedName("end") val end: Int,
    @SerializedName("itunesid") val itunesId: Int?,
    @SerializedName("art") val artUrl: String? = null,
    @SerializedName("original_artist") val originalArtist: String? = null,
    var videoId: String? = null
) : Parcelable

// File: java\com\example\holodex\data\model\HolodexVideoItem.kt
package com.example.holodex.data.model

import com.google.gson.annotations.SerializedName

data class HolodexVideoItem(
    @SerializedName("id") val id: String,
    @SerializedName("title") val title: String,
```

```
    @SerializedName("type") val type: String,
    @SerializedName("topic_id") val topicId: String?,
    @SerializedName("available_at") val availableAt: String,
    @SerializedName("published_at") val publishedAt: String?,
    @SerializedName("duration") val duration: Long,
    @SerializedName("status") val status: String,
    @SerializedName("channel") val channel: HolodexChannelMin,
    @SerializedName("songcount") val songcount: Int?,
    @SerializedName("description") val description: String?,

    // This field will be populated when fetching a single video with "include=songs"
    @SerializedName("songs") val songs: List<HolodexSong>? = null
)

// HolodexChannelMin remains the same
data class HolodexChannelMin(
    @SerializedName("id") val id: String?,
    @SerializedName("name") val name: String,
    @SerializedName("english_name") val englishName: String?,
    @SerializedName("org") var org: String?,
    @SerializedName("type") val type: String?,
    @SerializedName("photo") val photoUrl: String?
)

// File: java\com\example\holodex\data\model\PaginatedVideosResponse.kt
package com.example.holodex.data.model

import com.google.gson.annotations.SerializedName

data class PaginatedVideosResponse(
    @SerializedName("total") val total: String?, // API spec says number, but examples sometim
    @SerializedName("items") val items: List<HolodexVideoItem>
) {
    // Convenience getter for total as Int
    fun getTotalAsInt(): Int? {
        return total?.toIntOrNull()
    }
}

// File: java\com\example\holodex\data\model\VideoSearchRequest.kt
package com.example.holodex.data.model

import com.google.gson.annotations.SerializedName

data class VideoSearchRequest(
    @SerializedName("sort") val sort: String = "newest",
    @SerializedName("lang") val lang: List<String>? = null,
    @SerializedName("target") val target: List<String>, // e.g., ["stream", "clip"]
    @SerializedName("conditions") val conditions: List<SearchCondition>? = null, // For text s
    @SerializedName("topic") val topic: List<String>? = null, // For topic filtering
    @SerializedName("vch") val vch: List<String>? = null, // For channel ID search
    @SerializedName("org") val org: List<String>? = null, // For organization filtering
    @SerializedName("paginated") val paginated: Boolean = true,
    @SerializedName("offset") val offset: Int = 0,
    @SerializedName("limit") val limit: Int = 25,
```

```
    // @SerializedName("status") val status: List<String>? = null // REMOVE if /search/videoSe
    // 'comment' field was also in the original openapi spec example but not in your data clas
    @SerializedName("comment") val comment: List<String>? = null
)

data class SearchCondition(
    @SerializedName("text") val text: String
)

// File: java\com\example\holodex\data\model\discovery\ChannelDetails.kt
package com.example.holodex.data.model.discovery

import com.google.gson.annotations.SerializedName

/**
 * Represents the full details of a channel from the /channels/{id} endpoint.
 */
data class ChannelDetails(
    @SerializedName("id") val id: String,
    @SerializedName("name") val name: String,
    @SerializedName("english_name") val englishName: String?,
    @SerializedName("description") val description: String?,
    @SerializedName("photo") val photoUrl: String?,
    @SerializedName("banner") val bannerUrl: String?,
    @SerializedName("org") val org: String?,
    @SerializedName("suborg") val suborg: String?,
    @SerializedName("twitter") val twitter: String?,
    @SerializedName("group") val group: String? // Add the correct field for grouping
)

// File: java\com\example\holodex\data\model\discovery\DiscoveryResponse.kt
package com.example.holodex.data.model.discovery

import com.example.holodex.data.model.HolodexVideoItem
import com.google.gson.annotations.SerializedName


//Represents the entire aggregated response from the /musicdex/discovery/ endpoints.

data class DiscoveryResponse(
    @SerializedName("recentSingingStreams") val recentSingingStreams: List<SingingStreamShelfI
    @SerializedName("channels") val channels: List<DiscoveryChannel>?,
    @SerializedName("recommended") val recommended: RecommendedPlaylists?
)

data class SingingStreamShelfItem(
    @SerializedName("video") val video: HolodexVideoItem,
    // The playlist object here is a full playlist with content
    @SerializedName("playlist") val playlist: FullPlaylist
)

data class DiscoveryChannel(
    @SerializedName("id") val id: String,
    @SerializedName("name") val name: String,
    @SerializedName("english_name") val englishName: String?,
```

```
    @SerializedName("photo") val photoUrl: String?,
    @SerializedName("song_count") val songCount: Int?,
    @SerializedName("suborg") val suborg: String? // Add the missing property
)

data class RecommendedPlaylists(
    @SerializedName("playlists") val playlists: List<PlaylistStub>
)

// File: java\com\example\holodex\data\model\discovery\FullPlaylist.kt
// File: java/com/example/holodex/data/model/discovery/FullPlaylist.kt
package com.example.holodex.data.model.discovery

import com.google.gson.annotations.SerializedName

data class FullPlaylist(
    @SerializedName("id") val id: String,
    @SerializedName("title") val title: String,
    @SerializedName("description") val description: String?,
    @SerializedName("type") val type: String?,
    @SerializedName("created_at") val createdAt: String?,
    @SerializedName("updated_at") val updatedAt: String?,
    @SerializedName("content") val content: List<MusicdexSong>?
)

// File: java\com\example\holodex\data\model\discovery\MusicdexSong.kt
package com.example.holodex.data.model.discovery

import com.google.gson.annotations.SerializedName

data class MusicdexSong(
    @SerializedName("id") val id: String?,
    @SerializedName("song_id") val songId: String,
    @SerializedName("name") val name: String,
    @SerializedName("original_artist") val originalArtist: String?,
    @SerializedName("art") val artUrl: String?,
    @SerializedName("video_id") val videoId: String,
    @SerializedName("start") val start: Int,
    @SerializedName("end") val end: Int,
    @SerializedName("available_at") val available_at: String?,

    @SerializedName("channel_id") val channelId: String?,
    @SerializedName("channel") val channel: MusicdexChannel,
    @SerializedName("ts") val ts: String? = null
)

data class MusicdexChannel(
    @SerializedName("id") val id: String?,
    @SerializedName("name") val name: String,
    @SerializedName("english_name") val englishName: String?,
    @SerializedName("photo") val photoUrl: String?,
    @SerializedName("suborg") val suborg: String?
)

// File: java\com\example\holodex\data\model\discovery\PlaylistStub.kt
```

```
package com.example.holodex.data.model.discovery

import com.google.gson.annotations.SerializedName

/**
 * Represents a playlist "stub" as returned in discovery carousels.
 * It contains metadata but not the full list of songs.
 */
data class PlaylistStub(
    @SerializedName("id") val id: String,
    @SerializedName("title") val title: String,
    @SerializedName("type") val type: String, // e.g., "ugp", "radio/artist"
    @SerializedName("art_context") val artContext: ArtContext?,
    @SerializedName("description") val description: String?
)

data class ArtContext(
    @SerializedName("videos") val videos: List<String>?, // Changed name and type
    @SerializedName("channels") val channels: List<String>?,
    @SerializedName("channel_photo") val channelPhotoUrl: String?
)

// File: java\com\example\holodex\data\repository\DownloadRepository.kt
package com.example.holodex.data.repository

import android.content.Context
import android.net.Uri
import android.os.Build
import android.os.Environment
import android.provider.MediaStore
import androidx.annotation.OptIn
import androidx.core.net.toUri
import androidx.media3.common.MediaItem
import androidx.media3.common.util.UnstableApi
import androidx.media3.datasource.DataSource
import androidx.media3.datasource.cache.CacheDataSource
import androidx.media3.datasource.cache.SimpleCache
import androidx.media3.exoplayer.offline.Download
import androidx.media3.exoplayer.offline.DownloadHelper
import androidx.media3.exoplayer.offline.DownloadManager
import androidx.media3.exoplayer.offline.DownloadRequest
import androidx.media3.exoplayer.offline.DownloadService
import androidx.media3.exoplayer.scheduler.Requirements
import androidx.work.Data
import androidx.work.ExistingWorkPolicy
import androidx.work.OneTimeWorkRequestBuilder
import androidx.work.WorkManager
import com.example.holodex.background.M4AExportWorker
import com.example.holodex.data.db.DownloadStatus
import com.example.holodex.data.db.DownloadedItemDao
import com.example.holodex.data.db.DownloadedItemEntity
import com.example.holodex.data.db.LikedItemEntity
import com.example.holodex.data.db.ParentVideoMetadataDao
import com.example.holodex.data.db.ParentVideoMetadataEntity
import com.example.holodex.data.model.HolodexSong
```

```
import com.example.holodex.data.model.HolodexVideoItem
import com.example.holodex.data.repository.DownloadRepository.DownloadCompletedEvent
import com.example.holodex.di.ApplicationScope
import com.example.holodex.di.DownloadCache
import com.example.holodex.di.UpstreamDataSource
import com.example.holodex.service.HolodexDownloadService
import com.example.holodex.util.ThumbnailQuality
import com.example.holodex.util.getYouTubeThumbnailUrl
import dagger.hilt.android.qualifiers.ApplicationContext
import kotlinx.coroutines.CancellationException
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.MutableSharedFlow
import kotlinx.coroutines.flow.SharedFlow
import kotlinx.coroutines.flow.asSharedFlow
import kotlinx.coroutines.flow.first
import kotlinx.coroutines.launch
import kotlinx.coroutines.suspendCancellableCoroutine
import kotlinx.coroutines.withContext
import kotlinx.coroutines.withTimeout
import org.jaudiotagger.audio.AudioFileIO
import org.jaudiotagger.tag.FieldKey
import timber.log.Timber
import java.io.File
import java.io.IOException
import java.nio.charset.StandardCharsets
import javax.inject.Inject
import javax.inject.Singleton
import kotlin.coroutines.resume
import kotlin.coroutines.resumeWithException

@UnstableApi
interface DownloadRepository {
    suspend fun startDownload(video: HolodexVideoItem, song: HolodexSong)
    suspend fun cancelDownload(itemId: String)
    suspend fun deleteDownloadById(itemId: String)
    fun getAllDownloads(): Flow<List<DownloadedItemEntity>>
    fun getDownloadById(itemId: String): Flow<DownloadedItemEntity?>
    suspend fun reconcileAllDownloads()
    suspend fun resumeDownload(itemId: String)
    suspend fun retryExportForItem(item: DownloadedItemEntity)
    suspend fun rescanStorageForDownloads()
    val downloadCompletedEvents: SharedFlow<DownloadCompletedEvent>
    suspend fun postDownloadCompletedEvent(event: DownloadCompletedEvent)

    data class DownloadCompletedEvent(val itemId: String, val localFileUri: String)
}

@UnstableApi
@Singleton
@OptIn(UnstableApi::class)
class DownloadRepositoryImpl @Inject constructor(
    @ApplicationContext private val context: Context,
    private val downloadedItemDao: DownloadedItemDao,
```

```
    private val youtubeStreamRepository: YouTubeStreamRepository,
    @DownloadCache private val downloadCache: SimpleCache,
    @UpstreamDataSource private val upstreamDataSourceFactory: DataSource.Factory,
    private val parentVideoMetadataDao: ParentVideoMetadataDao,
    private val media3DownloadManager: DownloadManager,
    @ApplicationScope private val applicationScope: CoroutineScope,
    private val holodexRepository: HolodexRepository,
    private val workManager: WorkManager
) : DownloadRepository {
    companion object {
        private const val TAG = "DownloadRepositoryImpl"
    }

    private val _downloadCompletedEvents = MutableSharedFlow<DownloadCompletedEvent>()
    override val downloadCompletedEvents: SharedFlow<DownloadCompletedEvent> =
        _downloadCompletedEvents.asSharedFlow()

    override suspend fun startDownload(video: HolodexVideoItem, song: HolodexSong) {
        val itemId = LikedItemEntity.generateSongItemId(video.id, song.start)
        val displayTitle = song.name.ifBlank { video.title }
        val durationSec = (song.end - song.start).toLong()

        val existing = downloadedItemDao.getById(itemId)
        if (existing?.downloadStatus in listOf(
                DownloadStatus.ENQUEUED,
                DownloadStatus.DOWNLOADING,
                DownloadStatus.COMPLETED
            )
        ) {
            Timber.w("$TAG: Download for $itemId is already in progress or completed. Skipping
            return
        }

        Timber.d("$TAG: Initiating download process for item: $itemId ('$displayTitle')")

        // The heavy lifting (network calls) is offloaded to the application scope
        // to ensure it survives ViewModel lifecycle changes.
        applicationScope.launch(Dispatchers.IO) {
            try {
                // IMPROVEMENT: Resolve stream and determine format dynamically
                Timber.d("$TAG: Resolving stream for $itemId...")
                val streamDetails = withTimeout(30_000) {
                    youtubeStreamRepository.getAudioStreamDetails(video.id).getOrThrow()
                }
                val streamUri = streamDetails.streamUrl.toUri()

                val targetFormat = "M4A"
                Timber.i("$TAG: Determined target format for $itemId: $targetFormat (source wa

                // IMPROVEMENT: Use DownloadHelper for efficient partial downloading
                val clipStartTimeMs = song.start * 1000L
                val clipDurationMs = durationSec * 1000L

                val cacheDataSourceFactory = CacheDataSource.Factory()
                    .setCache(downloadCache)
```

```
    .setUpstreamDataSourceFactory(upstreamDataSourceFactory)

val downloadHelperFactory =
    DownloadHelper.Factory().setDataSourceFactory(cacheDataSourceFactory)
val mediaItemForHelper = MediaItem.Builder().setUri(streamUri).build()
val downloadHelper = downloadHelperFactory.create(mediaItemForHelper)

val downloadRequest = suspendCancellableCoroutine<DownloadRequest> { continuat
    downloadHelper.prepare(object : DownloadHelper.Callback {
        override fun onPrepared(
            helper: DownloadHelper,
            tracksInfoAvailable: Boolean
        ) {
            try {
                val request = helper.getDownloadRequest(
                    itemId,
                    displayTitle.toByteArray(StandardCharsets.UTF_8),
                    clipStartTimeMs,
                    clipDurationMs
                )
                Timber.d("$TAG: DownloadRequest prepared with byte range. ID:
                continuation.resume(request)
            } catch (e: Exception) {
                continuation.resumeWithException(e)
            }
        }

        override fun onPrepareError(helper: DownloadHelper, e: IOException) {
            continuation.resumeWithException(e)
        }
    })
    continuation.invokeOnCancellation { downloadHelper.release() }
}
downloadHelper.release()

try {
    Timber.d("$TAG: Proactively caching full video details for ${video.id} bef
    // Force a network refresh to ensure we get the latest song list.
    // This populates the VideoDao, which is now our source of truth.
    holodexRepository.getVideoWithSongs(video.id, forceRefresh = true)
} catch (e: Exception) {
    // Log the error but DO NOT fail the download.
    // The download can still proceed, but the user might see an incomplete
    // song list offline until they go online again.
    Timber.e(
        e,
        "$TAG: Failed to proactively cache video details for ${video.id}. Down
    )
}

// IMPROVEMENT: Consolidate all metadata before enqueuing
val sortedSongs = video.songs?.sortedBy { it.start }
val trackNumber = sortedSongs?.indexOf(song)?.plus(1)

val entity = DownloadedItemEntity(
```

```
                videoId = itemId,
                title = displayTitle,
                artistText = video.channel.name, // Use channel name for Artist tag
                channelId = video.channel.id ?: "unknown",
                artworkUrl = song.artUrl ?: video.channel.photoUrl,
                durationSec = durationSec,
                localFileUri = null,
                downloadStatus = DownloadStatus.ENQUEUED,
                downloadedAt = null,
                downloadId = null,
                progress = 0,
                trackNumber = trackNumber,
                fileName = "", // Will be set by worker
                targetFormat = targetFormat // Save the detected format
            )
            downloadedItemDao.insertOrUpdate(entity)

            // Save parent metadata for the worker to retrieve album title
            val parentMetadata = ParentVideoMetadataEntity(
                videoId = video.id,
                title = video.title,
                channelName = video.channel.name,
                channelId = video.channel.id ?: "unknown",
                thumbnailUrl = getYouTubeThumbnailUrl(
                    video.id,
                    ThumbnailQuality.HIGH
                ).firstOrNull(),
                description = video.description,
                totalDurationSec = video.duration
            )
            parentVideoMetadataDao.insert(parentMetadata)

            // Enqueue the download with Media3's service
            val requirements =
                Requirements(Requirements.NETWORK) // Or make this user-configurable
            DownloadService.sendSetRequirements(
                context,
                HolodexDownloadService::class.java,
                requirements,
                true
            )
            DownloadService.sendAddDownload(
                context,
                HolodexDownloadService::class.java,
                downloadRequest,
                true
            )

            Timber.i("$TAG: Successfully enqueued download for item ID: $itemId")

        } catch (e: CancellationException) {
            Timber.w("$TAG: Download setup was cancelled for $itemId")
            downloadedItemDao.updateStatus(itemId, DownloadStatus.FAILED)
        } catch (e: Exception) {
            Timber.e(e, "$TAG: Critical failure during download setup for $itemId.")
```

```
                downloadedItemDao.updateStatus(itemId, DownloadStatus.FAILED)
            }
        }
    }

    override suspend fun resumeDownload(itemId: String) {
        withContext(Dispatchers.IO) {
            Timber.d("DownloadRepository: Attempting to resume download for item ID: $itemId")

            val existingItem = downloadedItemDao.getById(itemId)
            if (existingItem?.downloadStatus == DownloadStatus.PAUSED) {
                // Update status to enqueued and let the download manager handle it
                downloadedItemDao.updateStatus(itemId, DownloadStatus.ENQUEUED)

                // Send resume command to download service with foreground parameter
                DownloadService.sendResumeDownloads(
                    context,
                    HolodexDownloadService::class.java,
                    true
                )

                Timber.i("DownloadRepository: Resume command sent for item ID: $itemId")
            } else {
                Timber.w("DownloadRepository: Cannot resume download for $itemId - current sta
            }
        }
    }

    override suspend fun cancelDownload(itemId: String) {
        Timber.d("DownloadRepository: Attempting to cancel download for item ID: $itemId")

        WorkManager.getInstance(context).cancelUniqueWork("export_$itemId")

        DownloadService.sendRemoveDownload(
            context,
            HolodexDownloadService::class.java,
            itemId,
            false
        )
    }

    override suspend fun deleteDownloadById(itemId: String) {
        withContext(Dispatchers.IO) {
            Timber.i("DownloadRepository: Starting robust delete for item ID: $itemId")

            val itemToDelete = downloadedItemDao.getById(itemId) ?: run {
                Timber.w("DownloadRepository: Item $itemId not found in DB for deletion.")
                return@withContext
            }

            itemToDelete.localFileUri?.let { uriString ->
                val fileUri = uriString.toUri()
                var fileDeleted = false

                // IMPORTANT: Get the file path FIRST, before any deletion attempts
```

```
        val filePath = if (Build.VERSION.SDK_INT <= Build.VERSION_CODES.P) {
            Timber.d("DownloadRepository: Getting file path for API 28 fallback...")
            getPathFromUri(fileUri).also { path ->
                Timber.d("DownloadRepository: Retrieved path: $path")
            }
        } else null

        // 1. First try MediaStore deletion
        try {
            val deletedRows = context.contentResolver.delete(fileUri, null, null)
            if (deletedRows > 0) {
                Timber.i("DownloadRepository: Successfully deleted MediaStore entry fo
                fileDeleted = true
            } else {
                Timber.w("DownloadRepository: MediaStore deletion returned 0 rows for
            }
        } catch (e: Exception) {
            Timber.e(
                e,
                "DownloadRepository: MediaStore deletion failed for URI: $uriString"
            )
        }

        // 2. Then try direct file deletion for API 28 and below
        if (Build.VERSION.SDK_INT <= Build.VERSION_CODES.P && !filePath.isNullOrBlank(
            try {
                val file = File(filePath)
                if (file.exists()) {
                    if (file.delete()) {
                        Timber.i("DownloadRepository: (API 28) Successfully deleted fi
                        fileDeleted = true
                    } else {
                        Timber.e("DownloadRepository: (API 28) Failed to delete file a
                    }
                } else {
                    Timber.w("DownloadRepository: (API 28) File doesn't exist at: $fil
                }
            } catch (e: Exception) {
                Timber.e(
                    e,
                    "DownloadRepository: (API 28) Error during file deletion at: $file
                )
            }
        }

        if (!fileDeleted) {
            Timber.w("DownloadRepository: Failed to confirm file deletion for: $itemId
        }
    } ?: run {
        Timber.w("DownloadRepository: No localFileUri found for item: $itemId")
    }

    // Always clean up the app state, even if file deletion failed
    try {
        DownloadService.sendRemoveDownload(
```

```
                context,
                HolodexDownloadService::class.java,
                itemId,
                false
            )
            Timber.d("DownloadRepository: Sent remove download command")
        } catch (e: Exception) {
            Timber.e(e, "DownloadRepository: Failed to send remove download command")
        }

        try {
            downloadCache.removeResource(itemId)
            Timber.d("DownloadRepository: Removed from download cache")
        } catch (e: Exception) {
            Timber.e(e, "DownloadRepository: Failed to remove from download cache")
        }

        try {
            downloadedItemDao.deleteById(itemId)
            Timber.d("DownloadRepository: Deleted from database")
        } catch (e: Exception) {
            Timber.e(e, "DownloadRepository: Failed to delete from database")
        }

        Timber.i("DownloadRepository: Deletion process complete for item: $itemId")
    }
}

override suspend fun postDownloadCompletedEvent(event: DownloadCompletedEvent) {
    _downloadCompletedEvents.emit(event)
}

private fun getPathFromUri(uri: Uri): String? {
    var path: String? = null
    val projection = arrayOf(MediaStore.Audio.Media.DATA)
    try {
        context.contentResolver.query(uri, projection, null, null, null)?.use { cursor ->
            if (cursor.moveToFirst()) {
                val columnIndex = cursor.getColumnIndexOrThrow(MediaStore.Audio.Media.DATA
                path = cursor.getString(columnIndex)
                Timber.d("DownloadRepository: Successfully retrieved path from URI: $path"
            } else {
                Timber.w("DownloadRepository: Cursor empty for URI: $uri")
            }
        }
    } catch (e: Exception) {
        Timber.e(e, "DownloadRepository: Failed to get path from URI: $uri")
    }
    return path
}

override suspend fun retryExportForItem(item: DownloadedItemEntity) {
    if (item.downloadStatus != DownloadStatus.EXPORT_FAILED) {
        Timber.w("Cannot retry export for ${item.videoId}, status is not EXPORT_FAILED.")
        return
```

```
    }

    Timber.i("Retrying export for ${item.videoId}")

    // This simulates the logic from DownloadCompletionObserver to re-enqueue the worker
    val parentVideoId = item.videoId.split('_').firstOrNull()
    val parentMetadata = parentVideoId?.let { parentVideoMetadataDao.getById(it) }

    if (parentMetadata == null) {
        Timber.e("Cannot retry export for ${item.videoId}, parent metadata not found.")
        downloadedItemDao.updateStatus(
            item.videoId,
            DownloadStatus.FAILED
        ) // Mark as permanently failed
        return
    }

    val albumName = parentMetadata.title
    val startTimeSeconds = item.videoId.split('_').getOrNull(1)?.toLongOrNull() ?: 0L
    val clipStartMs = startTimeSeconds * 1000
    val clipEndMs = (startTimeSeconds + item.durationSec) * 1000

    // Re-use the cache URI which should still be valid
    val cacheUri = "cache://${item.videoId}"

    val workData = Data.Builder()
        .putString(M4AExportWorker.KEY_ITEM_ID, item.videoId)
        .putString(M4AExportWorker.KEY_ORIGINAL_URI, cacheUri)
        .putString(M4AExportWorker.KEY_SONG_TITLE, item.title)
        .putString(M4AExportWorker.KEY_ARTIST_NAME, item.artistText)
        .putString(M4AExportWorker.KEY_ALBUM_NAME, albumName)
        .putString(M4AExportWorker.KEY_ARTWORK_URI, item.artworkUrl)
        .putLong(M4AExportWorker.KEY_CLIP_START_MS, clipStartMs)
        .putLong(M4AExportWorker.KEY_CLIP_END_MS, clipEndMs)
        .apply {
            item.trackNumber?.let { putInt(M4AExportWorker.KEY_TRACK_NUMBER, it) }
        }
        .build()

    val exportRequest = OneTimeWorkRequestBuilder<M4AExportWorker>()
        .setInputData(workData)
        .build()

    workManager.enqueueUniqueWork(
        "export_${item.videoId}",
        ExistingWorkPolicy.REPLACE,
        exportRequest
    )
    downloadedItemDao.updateStatus(
        item.videoId,
        DownloadStatus.PROCESSING
    ) // Set status back to processing
}

override suspend fun reconcileAllDownloads() {
```

```
    withContext(Dispatchers.IO) {
        Timber.d("DownloadRepository: Starting full download reconciliation.")
        val appDbDownloads = downloadedItemDao.getAllDownloads().first()
        val media3ActiveDownloads = media3DownloadManager.currentDownloads
        val media3ActiveDownloadIds = media3ActiveDownloads.map { it.request.id }.toSet()

        for (item in appDbDownloads) {
            // Reconcile items stuck in active states
            if (item.downloadStatus == DownloadStatus.DOWNLOADING || item.downloadStatus =
                if (!media3ActiveDownloadIds.contains(item.videoId)) {
                    Timber.w("Reconcile: Item ${item.videoId} is stuck in an active state.
                    downloadedItemDao.updateStatus(item.videoId, DownloadStatus.FAILED)
                } else {
                    val media3Download =
                        media3ActiveDownloads.find { it.request.id == item.videoId }
                    if (media3Download?.state == Download.STATE_FAILED) {
                        Timber.w("Reconcile: Item ${item.videoId} is FAILED in Media3. Syn
                        downloadedItemDao.updateStatus(item.videoId, DownloadStatus.FAILED
                    }
                }
            }

            // Reconcile completed items with missing files
            if (item.downloadStatus == DownloadStatus.COMPLETED) {
                val uriString = item.localFileUri
                if (uriString.isNullOrBlank()) {
                    Timber.w("Reconcile: Item ${item.videoId} is COMPLETED but has no URI.
                    deleteDownloadById(item.videoId)
                    continue
                }
                val fileExists = try {
                    context.contentResolver.openAssetFileDescriptor(uriString.toUri(), "r"
                        ?.use { true } ?: false
                } catch (_: Exception) {
                    false
                }

                if (!fileExists) {
                    Timber.w("Reconcile: File for completed item ${item.videoId} is missin
                    deleteDownloadById(item.videoId)
                }
            }
        }
        Timber.i("DownloadRepository: Reconciliation complete.")
    }
}

override fun getAllDownloads(): Flow<List<DownloadedItemEntity>> =
    downloadedItemDao.getAllDownloads()

override fun getDownloadById(itemId: String): Flow<DownloadedItemEntity?> =
    downloadedItemDao.getDownloadByIdFlow(itemId)

override suspend fun rescanStorageForDownloads() {
    withContext(Dispatchers.IO) {
```

```
Timber.d("$TAG: Starting storage re-scan for orphaned downloads.")
val musicDir =
    Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_MUSIC)
val appMusicDir = File(musicDir, "HolodexMusic")

if (!appMusicDir.exists() || !appMusicDir.isDirectory) {
    Timber.d("$TAG: HolodexMusic directory does not exist. No scan needed.")
    return@withContext
}

val mediaFiles =
    appMusicDir.listFiles { _, name -> name.endsWith(".m4a") } ?: return@withConte
if (mediaFiles.isEmpty()) {
    Timber.d("$TAG: No .m4a files found in HolodexMusic directory.")
    return@withContext
}

Timber.d("$TAG: Found ${mediaFiles.size} potential files to scan.")
val allDbDownloads = downloadedItemDao.getAllDownloads().first()
val dbIds = allDbDownloads.map { it.videoId }.toSet()

var importedCount = 0
for (file in mediaFiles) {
    try {
        val audioFile = AudioFileIO.read(file)
        val tag = audioFile.tag
        val comment = tag?.getFirst(FieldKey.COMMENT)

        if (comment != null && comment.startsWith("holodex_item_id::")) {
            val itemId = comment.substringAfter("holodex_item_id::")

            if (!dbIds.contains(itemId)) {
                // This is an orphaned file we need to re-import
                val title = tag.getFirst(FieldKey.TITLE)
                val artist = tag.getFirst(FieldKey.ARTIST)
                val album = tag.getFirst(FieldKey.ALBUM)
                val trackNum = tag.getFirst(FieldKey.TRACK)?.toIntOrNull()

                val parentVideoId = itemId.split('_').first()
                val songStartSec = itemId.split('_').getOrNull(1)?.toLongOrNull()
                val durationSec = (audioFile.audioHeader.trackLength).toLong()

                // Reconstruct the entity from the metadata
                val entity = DownloadedItemEntity(
                    videoId = itemId,
                    title = title.ifEmpty { "Unknown Title" },
                    artistText = artist.ifEmpty { "Unknown Artist" },
                    channelId = "", // This data is lost, but not critical
                    artworkUrl = null, // This data is lost
                    durationSec = durationSec,
                    localFileUri = Uri.fromFile(file).toString(),
                    downloadStatus = DownloadStatus.COMPLETED,
                    downloadedAt = file.lastModified(),
                    fileName = file.name,
                    targetFormat = "M4A",
```

```
                                downloadId = null,
                                progress = 100,
                                trackNumber = trackNum
                            )
                            downloadedItemDao.insertOrUpdate(entity)
                            importedCount++
                            Timber.i("$TAG: Re-imported orphaned file: ${file.name} (ID: $item
                        }
                    }
                } catch (e: Exception) {
                    Timber.e(e, "Failed to read metadata for file: ${file.name}")
                }
            }
            if (importedCount > 0) {
                Timber.i("$TAG: Successfully re-imported $importedCount orphaned downloads.")
            } else {
                Timber.d("$TAG: Re-scan finished. No new orphaned files found.")
            }
        }
    }
}


// File: java\com\example\holodex\data\repository\HolodexRepository.kt
// File: java/com/example/holodex/data/repository/HolodexRepository.kt
package com.example.holodex.data.repository

import androidx.media3.common.util.UnstableApi
import androidx.room.withTransaction
import com.example.holodex.auth.TokenManager
import com.example.holodex.background.LogAction
import com.example.holodex.background.SyncLogger
import com.example.holodex.data.api.AuthenticatedMusicdexApiService
import com.example.holodex.data.api.HolodexApiService
import com.example.holodex.data.api.LatestSongsRequest
import com.example.holodex.data.api.LikeRequest
import com.example.holodex.data.api.LikedSongApiDto
import com.example.holodex.data.api.MusicdexApiService
import com.example.holodex.data.api.Organization
import com.example.holodex.data.api.PaginatedChannelsResponse
import com.example.holodex.data.api.PaginatedSongsResponse
import com.example.holodex.data.api.PatchOperation
import com.example.holodex.data.api.PlaylistListResponse
import com.example.holodex.data.api.PlaylistUpdateRequest
import com.example.holodex.data.api.StarPlaylistRequest
import com.example.holodex.data.cache.BrowseCacheKey
import com.example.holodex.data.cache.BrowseListCache
import com.example.holodex.data.cache.CacheException
import com.example.holodex.data.cache.CachePolicy
import com.example.holodex.data.cache.FetcherResult
import com.example.holodex.data.cache.SearchCacheKey
import com.example.holodex.data.cache.SearchListCache
import com.example.holodex.data.db.AppDatabase
import com.example.holodex.data.db.CachedDiscoveryResponse
import com.example.holodex.data.db.DiscoveryDao
```

```
import com.example.holodex.data.db.FavoriteChannelDao
import com.example.holodex.data.db.FavoriteChannelEntity
import com.example.holodex.data.db.HistoryDao
import com.example.holodex.data.db.HistoryItemEntity
import com.example.holodex.data.db.LikedItemDao
import com.example.holodex.data.db.LikedItemEntity
import com.example.holodex.data.db.LikedItemType
import com.example.holodex.data.db.PlaylistDao
import com.example.holodex.data.db.PlaylistEntity
import com.example.holodex.data.db.PlaylistItemEntity
import com.example.holodex.data.db.StarredPlaylistDao
import com.example.holodex.data.db.StarredPlaylistEntity
import com.example.holodex.data.db.SyncMetadataDao
import com.example.holodex.data.db.SyncStatus
import com.example.holodex.data.db.VideoDao
import com.example.holodex.data.db.mappers.toEntity
import com.example.holodex.data.db.mappers.toFavoriteChannelEntity
import com.example.holodex.data.db.mappers.toLikedItemEntityShell
import com.example.holodex.data.db.toEntity
import com.example.holodex.data.model.HolodexChannelMin
import com.example.holodex.data.model.HolodexSong
import com.example.holodex.data.model.HolodexVideoItem
import com.example.holodex.data.model.SearchCondition
import com.example.holodex.data.model.VideoSearchRequest
import com.example.holodex.data.model.discovery.ChannelDetails
import com.example.holodex.data.model.discovery.DiscoveryResponse
import com.example.holodex.data.model.discovery.FullPlaylist
import com.example.holodex.data.model.discovery.MusicdexSong
import com.example.holodex.data.model.discovery.PlaylistStub
import com.example.holodex.di.ApplicationScope
import com.example.holodex.playback.domain.model.PlaybackItem
import com.example.holodex.util.VideoFilteringUtil
import com.example.holodex.viewmodel.VideoListViewModel
import com.example.holodex.viewmodel.state.BrowseFilterState
import com.example.holodex.viewmodel.state.SongSegmentFilterMode
import com.example.holodex.viewmodel.state.ViewTypePreset
import kotlinx.coroutines.CoroutineDispatcher
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.async
import kotlinx.coroutines.awaitAll
import kotlinx.coroutines.coroutineScope
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.SharingStarted
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.flow.first
import kotlinx.coroutines.flow.map
import kotlinx.coroutines.flow.stateIn
import kotlinx.coroutines.launch
import kotlinx.coroutines.sync.Mutex
import kotlinx.coroutines.sync.withLock
import kotlinx.coroutines.withContext
import org.schabi.newpipe.extractor.NewPipe
```

```
import org.schabi.newpipe.extractor.ServiceList
import org.schabi.newpipe.extractor.StreamingService
import org.schabi.newpipe.extractor.stream.StreamInfoItem
import timber.log.Timber
import java.io.IOException
import java.net.URLEncoder
import java.nio.charset.StandardCharsets
import java.time.Instant
import java.util.concurrent.TimeUnit
import javax.inject.Inject
import javax.inject.Singleton

@UnstableApi
@Singleton
class HolodexRepository @Inject constructor(
    val holodexApiService: HolodexApiService,
    private val musicdexApiService: MusicdexApiService,
    private val authenticatedMusicdexApiService: AuthenticatedMusicdexApiService,
    private val discoveryDao: DiscoveryDao,
    private val browseListCache: BrowseListCache,
    private val searchListCache: SearchListCache,
    private val videoDao: VideoDao,
    private val likedItemDao: LikedItemDao,
    val playlistDao: PlaylistDao,
    private val appDatabase: AppDatabase,
    private val defaultDispatcher: CoroutineDispatcher,
    private val historyDao: HistoryDao,
    private val favoriteChannelDao: FavoriteChannelDao,
    internal val syncMetadataDao: SyncMetadataDao,
    private val starredPlaylistDao: StarredPlaylistDao,
    private val tokenManager: TokenManager,
    @ApplicationScope private val applicationScope: CoroutineScope
) {

    companion object {
        private const val TAG = "HolodexRepository"
        private val DISCOVERY_CACHE_TTL_MS = TimeUnit.HOURS.toMillis(1)
        const val DEFAULT_PAGE_SIZE = 50
        val DEFAULT_MUSIC_TOPICS =
            listOf("singing", "Music_Cover", "Original_Song", "3D_Stream")
        val CACHE_STALE_DURATION_MS = TimeUnit.HOURS.toMillis(1)
        private val TAG_SYNC = "SYNC_DEBUG"
    }

    private val browseNetworkMutex = Mutex()
    private val searchNetworkMutex = Mutex()
    private val videoDetailMutex = Mutex()

    val likedItemIds: StateFlow<Set<String>> =
        getObservableLikedItems()
            .map { likedItems -> likedItems.map { it.itemId }.toSet() }
            .stateIn(
                scope = applicationScope,
                started = SharingStarted.WhileSubscribed(5000L),
                initialValue = emptySet()
```

```
        )

    private val _availableOrganizations = MutableStateFlow<List<Pair<String, String?>>>(
        listOf("All Vtubers" to null, "Favorites" to "Favorites") // Initial default value
    )
    val availableOrganizations: StateFlow<List<Pair<String, String?>>> =
        _availableOrganizations.asStateFlow()

    init {
        // Fetch the dynamic organization list as soon as the repository is created.
        applicationScope.launch {
            fetchOrganizationList()
        }
    }

    private suspend fun fetchOrganizationList() {
        getOrganizationList() // This is the existing function that calls the API
            .onSuccess { orgs ->
                val orgList = orgs.map { org -> (org.name to org.name) }
                // Prepend the static options to the dynamic list
                _availableOrganizations.value =
                    listOf("All Vtubers" to null, "Favorites" to "Favorites") + orgList
            }
            .onFailure {
                Timber.e(it, "Failed to load dynamic organization list. Using hardcoded fallba
                // Populate with a fallback list on failure
                _availableOrganizations.value = listOf(
                    "All Vtubers" to null,
                    "Favorites" to "Favorites",
                    "Hololive" to "Hololive",
                    "Nijisanji" to "Nijisanji",
                    "Independents" to "Independents"
                )
            }
    }

    suspend fun fetchBrowseList(
        key: BrowseCacheKey,
        forceNetwork: Boolean = false,
        cachePolicy: CachePolicy = CachePolicy.CACHE_FIRST
    ): Result<FetcherResult<HolodexVideoItem>> = withContext(defaultDispatcher) {
        Timber.d("$TAG: fetchBrowseList called. Key: ${key.stringKey()}, forceNetwork: $forceN
        try {
            when (cachePolicy) {
                CachePolicy.CACHE_FIRST -> fetchBrowseWithCacheFirst(key, forceNetwork)
                CachePolicy.NETWORK_FIRST -> fetchBrowseWithNetworkFirst(key)
                CachePolicy.CACHE_ONLY -> fetchBrowseFromCacheOnly(key)
                CachePolicy.NETWORK_ONLY -> fetchBrowseFromNetworkOnly(key)
            }
        } catch (e: Exception) {
            Timber.e(e, "$TAG: Unhandled exception in fetchBrowseList for key ${key.stringKey(
            Result.failure(
                CacheException.StorageError(
                    "Failed to fetch browse list for key ${key.stringKey()}",
                    e
```

```
            )
        )
    }
}

private suspend fun fetchBrowseWithCacheFirst(
    key: BrowseCacheKey,
    forceNetwork: Boolean
): Result<FetcherResult<HolodexVideoItem>> {
    if (!forceNetwork) {
        browseListCache.get(key)?.let {
            Timber.d("$TAG: Browse CACHE_FIRST hit for key: ${key.stringKey()}")
            return Result.success(it)
        }
    }
    Timber.d("$TAG: Browse CACHE_FIRST miss or forceNetwork for key: ${key.stringKey()}. F
    return fetchBrowseFromNetworkWithFallback(key)
}

private suspend fun fetchBrowseWithNetworkFirst(key: BrowseCacheKey): Result<FetcherResult
    Timber.d("$TAG: Browse NETWORK_FIRST for key: ${key.stringKey()}. Fetching from networ
    return fetchBrowseFromNetworkWithFallback(key)
}

private suspend fun fetchBrowseFromCacheOnly(key: BrowseCacheKey): Result<FetcherResult<Ho
    return browseListCache.get(key)?.let {
        Timber.d("$TAG: Browse CACHE_ONLY hit for key: ${key.stringKey()}")
        Result.success(it)
    }
        ?: Result.failure(CacheException.NotFound("No cached browse data for key ${key.str
}

private suspend fun fetchBrowseFromNetworkOnly(key: BrowseCacheKey): Result<FetcherResult<
    Timber.d("$TAG: Browse NETWORK_ONLY for key: ${key.stringKey()}. Fetching directly fro
    return fetchBrowseFromNetwork(key)
}

private suspend fun fetchBrowseFromNetworkWithFallback(key: BrowseCacheKey): Result<Fetche
    val networkResult = fetchBrowseFromNetwork(key)
    if (networkResult.isSuccess) {
        return networkResult
    } else {
        val networkError = networkResult.exceptionOrNull() ?: CacheException.NetworkError(
            "Unknown browse network error for ${key.stringKey()}",
            null
        )
        Timber.w(
            networkError,
            "$TAG: Browse network fetch failed for ${key.stringKey()}. Trying stale cache.
        )
        browseListCache.getStale(key)?.let { staleData ->
            Timber.d("$TAG: Browse using STALE cache for ${key.stringKey()} after network
            return Result.success(staleData)
        } ?: return Result.failure(networkError)
    }
```

```
}

private suspend fun fetchBrowseFromNetwork(key: BrowseCacheKey): Result<FetcherResult<Holo
    return browseNetworkMutex.withLock {
        Timber.d("$TAG: Fetching BROWSE from network: Key=${key.stringKey()}")
        try {
            val apiRequest = VideoSearchRequest(
                sort = key.filters.sortField.apiValue,
                target = listOf("stream", "clip"),
                topic = key.filters.selectedPrimaryTopic?.let { listOf(it) }
                    ?: DEFAULT_MUSIC_TOPICS,
                org = key.filters.selectedOrganization?.let { listOf(it) },
                paginated = true,
                offset = key.pageOffset,
                limit = DEFAULT_PAGE_SIZE
            )
            val response = holodexApiService.searchVideosAdvanced(apiRequest)
            if (!response.isSuccessful || response.body() == null) {
                throw IOException("API Error (Browse) for ${key.filters.currentFilterDispl
            }

            val videosFromApi = response.body()!!.items
            var musicallyRelevantVideos =
                videosFromApi.filter { VideoFilteringUtil.isMusicContent(it) }

            if (key.filters.selectedViewPreset == ViewTypePreset.LATEST_STREAMS) {
                musicallyRelevantVideos = when (key.filters.songSegmentFilterMode) {
                    SongSegmentFilterMode.REQUIRE_SONGS -> musicallyRelevantVideos.filter
                        (it.songcount ?: 0) > 0 || !it.songs.isNullOrEmpty()
                    }

                    SongSegmentFilterMode.EXCLUDE_SONGS -> musicallyRelevantVideos.filter
                    SongSegmentFilterMode.ALL -> musicallyRelevantVideos
                }
            }
            Timber.d(
                "$TAG: Browse network fetch successful for ${key.stringKey()}. Items: ${mu
                    response.body()?.getTotalAsInt()
                }"
            )
            val fetcherResult = FetcherResult(
                musicallyRelevantVideos,
                response.body()?.getTotalAsInt(),
                key.pageOffset + musicallyRelevantVideos.size
            )
            browseListCache.store(key, fetcherResult)
            Result.success(fetcherResult)
        } catch (e: Exception) {
            Timber.e(
                e,
                "$TAG: Exception during browse network fetch for key ${key.stringKey()}"
            )
            Result.failure(
                CacheException.NetworkError(
                    "Browse network fetch failed for ${key.stringKey()}",
```

```
                    e
                )
            )
        }
    }
}

suspend fun fetchSearchList(
    key: SearchCacheKey,
    forceNetwork: Boolean = false,
    cachePolicy: CachePolicy = CachePolicy.CACHE_FIRST
): Result<FetcherResult<HolodexVideoItem>> = withContext(defaultDispatcher) {
    Timber.d("$TAG: fetchSearchList called. Key: ${key.stringKey()}, forceNetwork: $forceN
    try {
        when (cachePolicy) {
            CachePolicy.CACHE_FIRST -> fetchSearchWithCacheFirst(key, forceNetwork)
            CachePolicy.NETWORK_FIRST -> fetchSearchWithNetworkFirst(key)
            CachePolicy.CACHE_ONLY -> fetchSearchFromCacheOnly(key)
            CachePolicy.NETWORK_ONLY -> fetchSearchFromNetworkOnly(key)
        }
    } catch (e: Exception) {
        Timber.e(e, "$TAG: Unhandled exception in fetchSearchList for key ${key.stringKey(
        Result.failure(
            CacheException.StorageError(
                "Failed to fetch search list for key ${key.stringKey()}",
                e
            )
        )
    }
}

private suspend fun fetchSearchWithCacheFirst(
    key: SearchCacheKey,
    forceNetwork: Boolean
): Result<FetcherResult<HolodexVideoItem>> {
    if (!forceNetwork) {
        searchListCache.get(key)?.let {
            Timber.d("$TAG: Search CACHE_FIRST hit for key: ${key.stringKey()}")
            return Result.success(it)
        }
    }
    Timber.d("$TAG: Search CACHE_FIRST miss or forceNetwork for key: ${key.stringKey()}. F
    return fetchSearchFromNetworkWithFallback(key)
}

private suspend fun fetchSearchWithNetworkFirst(key: SearchCacheKey): Result<FetcherResult
    Timber.d("$TAG: Search NETWORK_FIRST for key: ${key.stringKey()}. Fetching from networ
    return fetchSearchFromNetworkWithFallback(key)
}

private suspend fun fetchSearchFromCacheOnly(key: SearchCacheKey): Result<FetcherResult<Ho
    return searchListCache.get(key)?.let {
        Timber.d("$TAG: Search CACHE_ONLY hit for key: ${key.stringKey()}")
        Result.success(it)
    }
```

```
        ?: Result.failure(CacheException.NotFound("No cached search data for key ${key.str
}

private suspend fun fetchSearchFromNetworkOnly(key: SearchCacheKey): Result<FetcherResult<
    Timber.d("$TAG: Search NETWORK_ONLY for key: ${key.stringKey()}. Fetching directly fro
    return fetchSearchFromNetwork(key)
}

private suspend fun fetchSearchFromNetworkWithFallback(key: SearchCacheKey): Result<Fetche
    val networkResult = fetchSearchFromNetwork(key)
    if (networkResult.isSuccess) {
        return networkResult
    } else {
        val networkError = networkResult.exceptionOrNull() ?: CacheException.NetworkError(
            "Unknown search network error for ${key.stringKey()}",
            null
        )
        Timber.w(
            networkError,
            "$TAG: Search network fetch failed for ${key.stringKey()}. Trying stale cache.
        )
        searchListCache.getStale(key)?.let { staleData ->
            Timber.d("$TAG: Search using STALE cache for ${key.stringKey()} after network
            return Result.success(staleData)
        } ?: return Result.failure(networkError)
    }
}

fun getStarredPlaylistsFlow(): Flow<List<StarredPlaylistEntity>> {
    return starredPlaylistDao.getStarredPlaylists()
}

@UnstableApi
private suspend fun fetchSearchFromNetwork(key: SearchCacheKey): Result<FetcherResult<Holo
    return searchNetworkMutex.withLock {
        Timber.d("$TAG: Fetching SEARCH from network: Key=${key.stringKey()}")
        try {
            val actualTextSearchConditions: List<SearchCondition>?
            val actualChannelIdForVch: List<String>?

            if (key.query.startsWith(VideoListViewModel.CHANNEL_ID_SEARCH_PREFIX)) {
                val actualChannelId =
                    key.query.removePrefix(VideoListViewModel.CHANNEL_ID_SEARCH_PREFIX)
                actualChannelIdForVch =
                    if (actualChannelId.isNotBlank()) listOf(actualChannelId) else null
                actualTextSearchConditions = null
            } else {
                actualTextSearchConditions = listOf(SearchCondition(text = key.query))
                actualChannelIdForVch = null
            }

            val apiRequest = VideoSearchRequest(
                sort = "newest",
                target = listOf("stream", "clip"),
                conditions = actualTextSearchConditions,
```

```
                topic = DEFAULT_MUSIC_TOPICS,
                vch = actualChannelIdForVch,
                paginated = true,
                offset = key.pageOffset,
                limit = DEFAULT_PAGE_SIZE
            )
            val response = holodexApiService.searchVideosAdvanced(apiRequest)
            if (!response.isSuccessful || response.body() == null) {
                throw IOException("API Error (Search) for '${key.query}': ${response.code(
            }

            val videosFromApi = response.body()!!.items
            val musicallyRelevantVideos =
                videosFromApi.filter { VideoFilteringUtil.isMusicContent(it) }
            Timber.d(
                "$TAG: Search network fetch successful for ${key.stringKey()}. Items: ${mu
                    response.body()?.getTotalAsInt()
                }"
            )
            val fetcherResult = FetcherResult(
                musicallyRelevantVideos,
                response.body()?.getTotalAsInt(),
                key.pageOffset + musicallyRelevantVideos.size
            )
            searchListCache.store(key, fetcherResult)
            Result.success(fetcherResult)
        } catch (e: Exception) {
            Timber.e(
                e,
                "$TAG: Exception during search network fetch for key ${key.stringKey()}"
            )
            Result.failure(
                CacheException.NetworkError(
                    "Search network fetch failed for ${key.stringKey()}",
                    e
                )
            )
        }
    }
}

suspend fun getVideoWithSongs(
    videoId: String,
    forceRefresh: Boolean = false
): Result<HolodexVideoItem> = withContext(defaultDispatcher) {
    videoDetailMutex.withLock {
        if (!forceRefresh) {
            val cachedVideoWithSongs = videoDao.getVideoWithSongsOnce(videoId)
            if (cachedVideoWithSongs != null && System.currentTimeMillis() - cachedVideoWi
                Timber.d("$TAG: getVideoWithSongs (ID: $videoId) - Returning FRESH network
                return@withLock Result.success(cachedVideoWithSongs.toDomain())
            }
        }

        Timber.d("$TAG: getVideoWithSongs (ID: $videoId) - No suitable cached version foun
```

```
        try {
            val response = holodexApiService.getVideoWithSongs(
                videoId = videoId,
                include = "songs,live_info,description",
                lang = "en"
            )

            if (response.isSuccessful && response.body() != null) {
                val videoFromApi = response.body()!!
                videoFromApi.songs?.forEach { it.videoId = videoFromApi.id }

                val existingVideoEntity = videoDao.getVideoByIdOnce(videoId)
                val entityToSave = videoFromApi.toEntity(
                    queryKey = existingVideoEntity?.listQueryKey,
                    insertionOrder = existingVideoEntity?.insertionOrder ?: 0,
                    currentTimestamp = System.currentTimeMillis()
                )
                val songEntitiesToSave =
                    videoFromApi.songs?.map { it.toEntity(videoFromApi.id) } ?: emptyList(

                appDatabase.withTransaction {
                    videoDao.insertVideo(entityToSave)
                    videoDao.deleteSongsForVideo(videoFromApi.id)
                    if (songEntitiesToSave.isNotEmpty()) {
                        videoDao.insertSongs(songEntitiesToSave)
                    }
                }
                Result.success(videoFromApi)
            } else {
                val errorBody = response.errorBody()?.string() ?: "Unknown API error"
                Timber.e("$TAG: API Error ${response.code()} for getVideoWithSongs ($video
                Result.failure(IOException("API Error ${response.code()} for $videoId: $er
            }
        } catch (e: Exception) {
            Timber.e(e, "$TAG: Network Exception for getVideoWithSongs ($videoId)")
            val staleVideo = videoDao.getVideoWithSongsOnce(videoId)
            if (staleVideo != null) {
                Timber.w("$TAG: Network failed, but returning STALE cached data for $video
                Result.success(staleVideo.toDomain())
            } else {
                Result.failure(
                    CacheException.NetworkError(
                        "Network fetch failed for $videoId and no cache is available.",
                        e
                    )
                )
            }
        }
    }
}

suspend fun addLikedSongSegment(video: HolodexVideoItem, song: HolodexSong) =
    withContext(defaultDispatcher) {
        // This function now assumes the ViewModel has determined this is a valid, syncabl
        val itemId = LikedItemEntity.generateSongItemId(video.id, song.start)
```

```
        // Try to find the serverId. A network call might be needed if not present.
        val songFromServer = fetchVideoAndFindSong(video.id, song.start)?.second
        val serverId = songFromServer?.id

        if (serverId == null) {
            // If it fails, log an error but don't crash. The like will simply not be sync
            Timber.e("Could not resolve serverId for song segment to sync like: ${song.nam
            return@withContext
        }

        val entity = LikedItemEntity(
            itemId = itemId, videoId = video.id, itemType = LikedItemType.SONG_SEGMENT,
            serverId = serverId, titleSnapshot = song.name.ifBlank { video.title },
            artistTextSnapshot = song.originalArtist ?: video.channel.name,
            albumTextSnapshot = video.title, artworkUrlSnapshot = song.artUrl ?: video.cha
            descriptionSnapshot = video.description, channelIdSnapshot = video.channel.id
            durationSecSnapshot = (song.end - song.start).toLong().coerceAtLeast(1L),
            actualSongName = song.name.ifBlank { null }, actualSongArtist = song.originalA
            actualSongArtworkUrl = song.artUrl, songStartSeconds = song.start,
            songEndSeconds = song.end, syncStatus = SyncStatus.DIRTY
        )
        likedItemDao.insert(entity)
    }

suspend fun removeFavoriteChannel(channelId: String) = withContext(defaultDispatcher) {
    Timber.tag(TAG_SYNC)
        .d("[FAV_CHANNEL] Optimistic Update: Marking channel $channelId for deletion.")
    favoriteChannelDao.softDelete(channelId)
}

suspend fun removeLikedItem(itemId: String) = withContext(defaultDispatcher) {
    Timber.tag("SYNC_DEBUG").i("====== LIKE ACTION (Repository) ======")
    Timber.tag("SYNC_DEBUG").i("Preparing to mark item for deletion: $itemId")
    val itemToRemove = likedItemDao.getLikedItem(itemId)

    if (itemToRemove == null) {
        Timber.tag("SYNC_DEBUG").w("[REMOVE] Item $itemId not found in DB. No action taken
        return@withContext
    }


    Timber.tag("SYNC_DEBUG")
        .d("[REMOVE] Current status of item $itemId is ${itemToRemove.syncStatus}. Updatin
    likedItemDao.performMarkForDeletion(
        itemId = itemToRemove.itemId,
        timestamp = System.currentTimeMillis()
    )
}

suspend fun addFavoriteChannel(videoItem: HolodexVideoItem) = withContext(defaultDispatche
    val channel = videoItem.channel
    if (channel.id == null) return@withContext
    Timber.tag(TAG_SYNC).d("[FAV_CHANNEL] Optimistic Update: Liking channel ${channel.name
```

```
    val entity = FavoriteChannelEntity(
        id = channel.id,
        name = channel.name,
        englishName = channel.englishName,
        photoUrl = channel.photoUrl,
        org = channel.org,
        subscriberCount = null,
        twitter = null,
        syncStatus = SyncStatus.DIRTY
    )
    favoriteChannelDao.insert(entity)
}

fun getItemsForPlaylist(playlistId: Long): Flow<List<PlaylistItemEntity>> =
    playlistDao.getItemsForPlaylist(playlistId)

fun getObservableLikedItems(): Flow<List<LikedItemEntity>> =
    likedItemDao.getAllLikedItemsSortedByDate()

fun getObservableLikedSongSegments(): Flow<List<LikedItemEntity>> =
    likedItemDao.getLikedSongSegmentsSortedByDate()

fun getAllPlaylists(): Flow<List<PlaylistEntity>> = playlistDao.getAllPlaylists()
suspend fun getLastItemOrderInPlaylist(playlistId: Long): Int? =
    withContext(defaultDispatcher) { playlistDao.getLastItemOrder(playlistId) }

suspend fun getPlaylistById(playlistId: Long): PlaylistEntity? =
    withContext(defaultDispatcher) { playlistDao.getPlaylistById(playlistId) }


suspend fun clearAllCachedData() = withContext(Dispatchers.IO) {
    Timber.i("$TAG: Clearing temporary application caches.")
    browseListCache.clear()
    searchListCache.clear()
    appDatabase.withTransaction {
        videoDao.clearAllSongs()
        videoDao.clearAllVideos()
    }
    Timber.i("$TAG: Temporary application caches cleared from repository.")
}

suspend fun cleanupExpiredCacheEntries() = withContext(Dispatchers.IO) {
    Timber.d("$TAG: Cleaning up expired cache entries.")
    browseListCache.cleanupExpiredEntries()
    searchListCache.cleanupExpiredEntries()
}

fun getFavoritedVideosPaged(
    offset: Int,
    limit: Int
): Flow<List<LikedItemEntity>> {
    Timber.d("$TAG: Fetching paged favorited videos. Offset: $offset, Limit: $limit")
    return likedItemDao.getLikedItemsByTypePaged(LikedItemType.VIDEO.name, limit, offset)
}
```

```
fun addSongToHistory(item: PlaybackItem) {
    applicationScope.launch(defaultDispatcher) {
        // 1. Save to local DB immediately for instant UI update. This part is correct.
        val historyEntity = HistoryItemEntity(
            playedAtTimestamp = System.currentTimeMillis(),
            itemId = item.id,
            videoId = item.videoId,
            songStartSeconds = item.clipStartSec?.toInt() ?: 0,
            title = item.title,
            artistText = item.artistText,
            artworkUrl = item.artworkUri,
            durationSec = item.durationSec,
            channelId = item.channelId
        )
        historyDao.upsert(historyEntity)
        Timber.tag("HISTORY").d("Saved '${item.title}' to local history cache.")

        // 2. Fire-and-forget the network request using the CORRECT server ID.
        var songServerId = item.serverUuid

        // 2. Check if the serverUuid is a valid UUID. If not, we need to fetch it.
        // A simple check is to see if it contains an underscore. A real UUID won't.
        if (songServerId == null || songServerId.contains("_")) {
            Timber.tag("HISTORY")
                .w("PlaybackItem has a local-style ID ('${item.id}') as its serverUuid. At
            val songFromApi =
                fetchVideoAndFindSong(item.videoId, item.clipStartSec?.toInt() ?: 0)?.seco
            if (songFromApi?.id != null) {
                songServerId = songFromApi.id
                Timber.tag("HISTORY").i("Successfully fetched real server UUID: $songServe
            }
        }

        if (songServerId == null) {
            Timber.tag("HISTORY")
                .e("Cannot track song in history: Could not find or resolve a valid server
            return@launch
        }

        try {
            val response = authenticatedMusicdexApiService.trackSongInHistory(songServerId
            if (response.isSuccessful) {
                Timber.tag("HISTORY")
                    .i("Successfully tracked song '${item.title}' (Server ID: $songServerI
            } else {
                Timber.tag("HISTORY")
                    .w("Failed to track song '$songServerId' in server history. Code: ${re
            }
        } catch (e: Exception) {
            Timber.tag("HISTORY")
                .e(e, "Network error while tracking song '$songServerId' in history.")
        }
    }
}
```

```
fun getHistory(): Flow<List<HistoryItemEntity>> {
    return historyDao.getHistory()
}

fun getFavoriteChannels(): Flow<List<FavoriteChannelEntity>> {
    return favoriteChannelDao.getFavoriteChannels()
}

fun getFavoriteChannelIds(): Flow<List<String>> {
    return favoriteChannelDao.getFavoriteChannelIds()
}

suspend fun getFavoritesFeed(
    channelIds: List<String>,
    filters: BrowseFilterState,
    offset: Int
): Result<FetcherResult<HolodexVideoItem>> = withContext(defaultDispatcher) {
    if (channelIds.isEmpty()) {
        return@withContext Result.success(FetcherResult(emptyList(), 0))
    }

    try {
        // This function now ONLY fetches from the Holodex API for the given IDs.
        val allVideos = coroutineScope {
            channelIds.map { channelId ->
                async {
                    val request = VideoSearchRequest(
                        sort = filters.sortField.apiValue,
                        vch = listOf(channelId),
                        topic = filters.selectedPrimaryTopic?.let { listOf(it) }
                            ?: DEFAULT_MUSIC_TOPICS,
                        paginated = true,
                        offset = offset, // Note: offset is less effective here, we'll han
                        limit = 15, // Fetch a decent number from each channel
                        target = listOf("stream", "clip")
                    )
                    holodexApiService.searchVideosAdvanced(request).body()?.items ?: empty
                }
            }
        }.flatMap { it.await() }

        val segmentFilteredVideos = when (filters.songSegmentFilterMode) {
            SongSegmentFilterMode.REQUIRE_SONGS -> allVideos.filter { (it.songcount ?: 0)
            SongSegmentFilterMode.EXCLUDE_SONGS -> allVideos.filter { (it.songcount ?: 0)
            SongSegmentFilterMode.ALL -> allVideos
        }

        // Sorting will now be handled in the ViewModel after merging.
        val finalList = segmentFilteredVideos.distinctBy { it.id }

        Result.success(FetcherResult(finalList, totalAvailable = null))
    } catch (e: Exception) {
        Result.failure(e)
    }
}
```

```
suspend fun getUpcomingMusicPaginated(
    org: String?,
    offset: Int
): Result<FetcherResult<HolodexVideoItem>> {
    val filters = BrowseFilterState.create(
        preset = ViewTypePreset.UPCOMING_STREAMS,
        songFilterMode = SongSegmentFilterMode.ALL,
        organization = org,
    )
    val key = BrowseCacheKey(filters, offset)
    return fetchBrowseList(key, forceNetwork = true)
}

suspend fun getDiscoveryHubContent(org: String): Result<DiscoveryResponse> =
    withContext(defaultDispatcher) {
        val cacheKey = "discovery_org_$org"
        try {
            val cachedResponse = discoveryDao.getResponse(cacheKey)
            if (cachedResponse != null) {
                val isStale =
                    System.currentTimeMillis() - cachedResponse.timestamp > DISCOVERY_CACH
                Timber.d("$TAG: Discovery Hub cache HIT for key '$cacheKey'. Is stale: $is
                if (!isStale) {
                    return@withContext Result.success(cachedResponse.data)
                } else {
                    launch { fetchAndCacheDiscoveryContent(org) }
                    return@withContext Result.success(cachedResponse.data)
                }
            }
            Timber.d("$TAG: Discovery Hub cache MISS for key '$cacheKey'. Fetching from ne
            return@withContext fetchAndCacheDiscoveryContent(org)

        } catch (e: Exception) {
            Timber.e(e, "$TAG: Exception in getDiscoveryHubContent for org '$org'")
            Result.failure(e)
        }
    }

suspend fun getFavoritesHubContent(): Result<DiscoveryResponse> =
    withContext(defaultDispatcher) {
        val cacheKey = "discovery_favorites"
        try {
            val cachedResponse = discoveryDao.getResponse(cacheKey)
            if (cachedResponse != null) {
                val isStale =
                    System.currentTimeMillis() - cachedResponse.timestamp > DISCOVERY_CACH
                Timber.d("$TAG: Favorites Hub cache HIT for key '$cacheKey'. Is stale: $is
                if (!isStale) {
                    return@withContext Result.success(cachedResponse.data)
                } else {
                    launch { fetchAndCacheFavoritesContent() }
                    return@withContext Result.success(cachedResponse.data)
                }
            }
```

```
            Timber.d("$TAG: Favorites Hub cache MISS for key '$cacheKey'. Fetching from ne
            return@withContext fetchAndCacheFavoritesContent()

        } catch (e: Exception) {
            Timber.e(e, "$TAG: Exception in getFavoritesHubContent")
            Result.failure(e)
        }
    }

suspend fun getHotSongsForCarousel(org: String?): Result<List<MusicdexSong>> =
    withContext(defaultDispatcher) {
        try {
            val response = holodexApiService.getHotSongs(organization = org, channelId = n
            if (response.isSuccessful && response.body() != null) {
                Result.success(response.body()!!)
            } else {
                Result.failure(IOException("API Error fetching hot songs for org '$org': $
            }
        } catch (e: Exception) {
            Result.failure(e)
        }
    }

@JvmName("getHotSongsForCarouselByChannelId")
suspend fun getHotSongsForCarousel(channelId: String): Result<List<MusicdexSong>> =
    withContext(defaultDispatcher) {
        try {
            val response =
                holodexApiService.getHotSongs(organization = null, channelId = channelId)
            if (response.isSuccessful && response.body() != null) {
                Result.success(response.body()!!)
            } else {
                Result.failure(IOException("API Error fetching hot songs for channel '$cha
            }
        } catch (e: Exception) {
            Result.failure(e)
        }
    }

suspend fun getFullPlaylistContent(playlistId: String): Result<FullPlaylist> =
    withContext(defaultDispatcher) {
        try {
            Timber.d("$TAG: Fetching full playlist content from network for ID: $playlistI

            val isSystemPlaylist = playlistId.startsWith(":")

            val response = if (isSystemPlaylist) {
                // System playlists like :history and :video require authentication
                authenticatedMusicdexApiService.getPlaylistContent(playlistId)
            } else {
                musicdexApiService.getPlaylistContent(playlistId)
            }

            if (response.isSuccessful && response.body() != null) {
                Result.success(response.body()!!)
```

```
            } else {
                Result.failure(IOException("API Error fetching playlist content for '$play
            }
        } catch (e: Exception) {
            Result.failure(e)
        }
    }

    private suspend fun fetchAndCacheDiscoveryContent(org: String): Result<DiscoveryResponse>
        return try {
            val response = musicdexApiService.getDiscoveryForOrg(org)
            if (response.isSuccessful && response.body() != null) {
                val discoveryResponse = response.body()!!
                val cacheEntry = CachedDiscoveryResponse(
                    pageKey = "discovery_org_$org",
                    data = discoveryResponse
                )
                discoveryDao.insertResponse(cacheEntry)
                Timber.i("$TAG: Successfully fetched and cached discovery content for org '$or
                Result.success(discoveryResponse)
            } else {
                Result.failure(IOException("API Error fetching discovery content for org '$org
            }
        } catch (e: Exception) {
            Result.failure(e)
        }
    }

    suspend fun getDiscoveryForChannel(channelId: String): Result<DiscoveryResponse> =
        withContext(defaultDispatcher) {
            try {
                val response = musicdexApiService.getDiscoveryForChannel(channelId)
                if (response.isSuccessful && response.body() != null) {
                    Result.success(response.body()!!)
                } else {
                    Result.failure(IOException("API Error fetching channel discovery: ${respon
                }
            } catch (e: Exception) {
                Result.failure(e)
            }
        }

    private suspend fun fetchAndCacheFavoritesContent(): Result<DiscoveryResponse> {
        return try {
            val response = authenticatedMusicdexApiService.getDiscoveryForFavorites()
            if (response.isSuccessful && response.body() != null) {
                val discoveryResponse = response.body()!!
                val cacheEntry = CachedDiscoveryResponse(
                    pageKey = "discovery_favorites",
                    data = discoveryResponse
                )
                discoveryDao.insertResponse(cacheEntry)
                Timber.i("$TAG: Successfully fetched and cached favorites discovery content.")
                Result.success(discoveryResponse)
            } else {
```

```
            Result.failure(IOException("API Error fetching favorites discovery content: ${
        }
    } catch (e: Exception) {
        Result.failure(e)
    }
}

suspend fun getLatestSongsPaginated(
    offset: Int,
    limit: Int = 25
): Result<PaginatedSongsResponse> {
    return try {
        val request = LatestSongsRequest(offset = offset, limit = limit, paginated = true)
        val response = holodexApiService.getLatestSongs(request)
        if (response.isSuccessful && response.body() != null) {
            Result.success(response.body()!!)
        } else {
            Result.failure(IOException("API Error fetching latest songs: ${response.code()
        }
    } catch (e: Exception) {
        Timber.e(e, "$TAG: Failed to fetch latest songs.")
        Result.failure(e)
    }
}

suspend fun getOrgChannelsPaginated(
    org: String,
    offset: Int,
    limit: Int = 25
): Result<PaginatedChannelsResponse> { // The return type to the ViewModel remains the sam
    return try {
        val response = holodexApiService.getChannels(
            organization = org,
            offset = offset,
            limit = limit
        )
        if (response.isSuccessful && response.body() != null) {
            val channelsList = response.body()!!
            // Manually construct the PaginatedChannelsResponse object.
            // The 'total' will be null, but the view model already handles this.
            val paginatedResponse = PaginatedChannelsResponse(
                total = null, // The API doesn't provide a total in this format
                items = channelsList
            )
            Result.success(paginatedResponse)
        } else {
            Result.failure(IOException("API Error fetching org channels for '$org': ${resp
        }
    } catch (e: Exception) {
        Timber.e(e, "Failed to fetch org channels for: $org")
        Result.failure(e)
    }
}

suspend fun getRadioContent(radioId: String): Result<FullPlaylist> =
```

```
    withContext(defaultDispatcher) {
        try {
            Timber.d("$TAG: Fetching radio content from network for ID: $radioId")
            val response = musicdexApiService.getRadioContent(radioId)
            if (response.isSuccessful && response.body() != null) {
                Result.success(response.body()!!)
            } else {
                Result.failure(IOException("API Error fetching radio content for '$radioId
            }
        } catch (e: Exception) {
            Result.failure(e)
        }
    }

suspend fun getOrgPlaylistsPaginated(
    org: String,
    type: String,
    offset: Int,
    limit: Int = 25
): Result<PlaylistListResponse> {
    return try {
        val response = musicdexApiService.getOrgPlaylists(
            org = org.ifBlank { "All_Vtubers" },
            type = type,
            offset = offset,
            limit = limit
        )
        if (response.isSuccessful && response.body() != null) {
            Result.success(response.body()!!)
        } else {
            Result.failure(IOException("API Error fetching org playlists (type: $type): ${
        }
    } catch (e: Exception) {
        Timber.e(e, "$TAG: Failed to fetch org playlists for org: $org, type: $type")
        Result.failure(e)
    }
}

suspend fun getChannelDetails(channelId: String): Result<ChannelDetails> =
    withContext(defaultDispatcher) {
        try {
            val response = holodexApiService.getChannelDetails(channelId)
            if (response.isSuccessful && response.body() != null) {
                Result.success(response.body()!!)
            } else {
                Result.failure(IOException("API Error fetching channel details: ${response
            }
        } catch (e: Exception) {
            Result.failure(e)
        }
    }

suspend fun fetchVideoAndFindSong(
    videoId: String,
    startTime: Int
```

```
): Pair<HolodexVideoItem, MusicdexSong?>? {
    val sgpId = withContext(Dispatchers.IO) {
        URLEncoder.encode(":video[id=$videoId]", StandardCharsets.UTF_8.toString())
    }
    val result = musicdexApiService.getPlaylistContent(sgpId)
    if (result.isSuccessful && result.body() != null) {
        val fullPlaylist = result.body()!!
        val matchingSong = fullPlaylist.content?.find { it.start == startTime }

        val videoItemShell = HolodexVideoItem(
            id = videoId,
            title = fullPlaylist.title,
            description = fullPlaylist.description,
            type = "stream",
            topicId = null,
            availableAt = matchingSong?.available_at?.toString() ?: "",
            publishedAt = null,
            duration = 0,
            status = "past",
            channel = HolodexChannelMin(
                id = matchingSong?.channel?.id ?: matchingSong?.channelId,
                name = matchingSong?.channel?.name ?: "Unknown",
                englishName = matchingSong?.channel?.englishName,
                org = null,
                type = "vtuber",
                photoUrl = matchingSong?.channel?.photoUrl
            ),
            songcount = fullPlaylist.content?.size,
            songs = fullPlaylist.content?.map { it.toHolodexSong() }
        )
        return Pair(videoItemShell, matchingSong)
    }
    return null
}

// --- Likes ---
suspend fun performUpstreamLikesSync(logger: SyncLogger) {
    // --- 1. Process local items marked for deletion ---
    val pendingDeletes =
        likedItemDao.getUnsyncedItems().filter { it.syncStatus == SyncStatus.PENDING_DELET
    if (pendingDeletes.isNotEmpty()) {
        logger.info("  -> Sending ${pendingDeletes.size} delete requests to the server.")
    }

    for (item in pendingDeletes) {
        if (item.serverId == null) {
            // This was liked and unliked before syncing. It's safe to just delete locally
            likedItemDao.deleteByItemId(item.itemId)
            logger.logItemAction(
                LogAction.UPSTREAM_DELETE_SUCCESS,
                item.actualSongName,
                item.itemId.hashCode().toLong(),
                null,
                "Local-only item, deleted directly."
            )
```

```
            continue
        }

        val response =
            authenticatedMusicdexApiService.deleteLike(LikeRequest(song_id = item.serverId
        if (response.isSuccessful || response.code() == 404) {
            logger.logItemAction(
                LogAction.UPSTREAM_DELETE_SUCCESS,
                item.actualSongName,
                item.itemId.hashCode().toLong(),
                item.serverId,
                "DELETE request sent successfully."
            )
        } else {
            logger.logItemAction(
                LogAction.UPSTREAM_DELETE_FAILED,
                item.actualSongName,
                item.itemId.hashCode().toLong(),
                item.serverId,
                "DELETE request failed: ${response.code()}."
            )
        }
    }

    // --- 2. Process local items that were newly liked ---
    val dirtyItems =
        likedItemDao.getUnsyncedItems().filter { it.syncStatus == SyncStatus.DIRTY }
    if (dirtyItems.isNotEmpty()) {
        logger.info("  -> Sending ${dirtyItems.size} add requests to the server.")
    }

    for (item in dirtyItems) {
        if (item.serverId == null) {
            logger.logItemAction(
                LogAction.UPSTREAM_UPSERT_FAILED,
                item.actualSongName,
                item.itemId.hashCode().toLong(),
                null,
                "Cannot sync, serverId is null."
            )
            continue
        }

        val response =
            authenticatedMusicdexApiService.addLike(LikeRequest(song_id = item.serverId))
        if (response.isSuccessful) {
            logger.logItemAction(
                LogAction.UPSTREAM_UPSERT_SUCCESS,
                item.actualSongName,
                item.itemId.hashCode().toLong(),
                item.serverId,
                "ADD request sent successfully."
            )
        } else {
            logger.logItemAction(
```

```
                LogAction.UPSTREAM_UPSERT_FAILED,
                item.actualSongName,
                item.itemId.hashCode().toLong(),
                item.serverId,
                "ADD request failed: ${response.code()}."
            )
        }
    }
}

suspend fun getAllLocalLikes(): List<LikedItemEntity> =
    likedItemDao.getAllLikedItemsOnce()

suspend fun updateLikesBatch(items: List<LikedItemEntity>) =
    likedItemDao.insert(items)

suspend fun deleteLikesBatch(itemIds: List<String>) =
    itemIds.forEach { likedItemDao.deleteByItemId(it) }

suspend fun getRemoteLikes(): List<LikedSongApiDto> {
    val allRemoteLikeDtos = mutableListOf<LikedSongApiDto>()
    var currentPage = 1
    var totalPages: Int
    do {
        val response = authenticatedMusicdexApiService.getLikes(page = currentPage)
        if (!response.isSuccessful) throw IOException("Failed to fetch remote likes page $
        val body = response.body()!!
        allRemoteLikeDtos.addAll(body.content)
        totalPages = body.page_count
        currentPage++
    } while (currentPage <= totalPages)
    return allRemoteLikeDtos
}

suspend fun insertRemoteLikesAsSynced(remoteLikes: List<LikedSongApiDto>) =
    likedItemDao.upsert(remoteLikes.map { it.toLikedItemEntityShell() })

suspend fun getOrphanedDirtyLikes(): List<LikedItemEntity> {
    return likedItemDao.getOrphanedDirtyItems()
}

suspend fun updateLike(item: LikedItemEntity) {
    likedItemDao.insert(item)
}


// --- Playlists ---
suspend fun getLocalPlaylists(): List<PlaylistEntity> = playlistDao.getAllPlaylistsOnce()
suspend fun getLocalPlaylistsByStatus(status: SyncStatus): List<PlaylistEntity> =
    playlistDao.getUnsyncedPlaylists().filter { it.syncStatus == status }

suspend fun performUpstreamPlaylistDeletions(logger: SyncLogger) {
    val pendingDeletes = getLocalPlaylistsByStatus(SyncStatus.PENDING_DELETE)
    if (pendingDeletes.isNotEmpty()) logger.info("  Processing ${pendingDeletes.size} pend
    pendingDeletes.forEach { playlist ->
```

```
        if (playlist.serverId != null) {
            val response = authenticatedMusicdexApiService.deletePlaylist(playlist.serverI
            if (response.isSuccessful || response.code() == 404) {
                logger.logItemAction(
                    LogAction.UPSTREAM_DELETE_SUCCESS,
                    playlist.name,
                    playlist.playlistId,
                    playlist.serverId
                )
                playlistDao.deletePlaylist(playlist.playlistId)
            } else {
                logger.logItemAction(
                    LogAction.UPSTREAM_DELETE_FAILED,
                    playlist.name,
                    playlist.playlistId,
                    playlist.serverId,
                    "Code: ${response.code()}"
                )
            }
        } else {
            playlistDao.deletePlaylist(playlist.playlistId) // Local-only item
        }
    }
}

suspend fun performUpstreamPlaylistUpserts(logger: SyncLogger) {
    val dirtyPlaylists = getLocalPlaylistsByStatus(SyncStatus.DIRTY)
    val userId = tokenManager.getUserId()?.toLongOrNull() ?: return
    if (dirtyPlaylists.isNotEmpty()) logger.info("  Processing ${dirtyPlaylists.size} dirt

    dirtyPlaylists.forEach { playlist ->
        val songServerIds = getLocalPlaylistItemServerIds(playlist.playlistId)
        val requestDto = PlaylistUpdateRequest(
            id = playlist.serverId,
            owner = userId,
            title = playlist.name,
            description = playlist.description,
            content = songServerIds
        )
        val response = authenticatedMusicdexApiService.createOrUpdatePlaylist(requestDto)

        if (response.isSuccessful) {
            if (playlist.serverId != null) {
                logger.logItemAction(
                    LogAction.UPSTREAM_UPSERT_SUCCESS,
                    playlist.name,
                    playlist.playlistId,
                    playlist.serverId
                )
                val updatedEntity = playlist.copy(syncStatus = SyncStatus.SYNCED)
                playlistDao.updatePlaylist(updatedEntity)
            } else {
                val newServerPlaylist = response.body()?.firstOrNull()
                if (newServerPlaylist != null) {
                    logger.logItemAction(
```

```
                        LogAction.UPSTREAM_UPSERT_SUCCESS,
                        newServerPlaylist.title,
                        playlist.playlistId,
                        newServerPlaylist.id
                    )
                    val finalEntity = newServerPlaylist.toEntity().copy(
                        playlistId = playlist.playlistId, // Keep the original local ID
                        syncStatus = SyncStatus.SYNCED
                    )
                    playlistDao.updatePlaylist(finalEntity)
                } else {
                    logger.logItemAction(
                        LogAction.UPSTREAM_UPSERT_FAILED,
                        playlist.name,
                        playlist.playlistId,
                        null,
                        "Server returned success but no playlist data."
                    )
                }
            }
        } else {
            logger.logItemAction(
                LogAction.UPSTREAM_UPSERT_FAILED,
                playlist.name,
                playlist.playlistId,
                playlist.serverId,
                "Code: ${response.code()}"
            )
        }
    }
}

suspend fun getRemotePlaylists(): List<PlaylistEntity> {
    val dtoList = authenticatedMusicdexApiService.getMyPlaylists().body() ?: emptyList()
    return dtoList.map { it.toEntity() } // Map the whole list
}

suspend fun insertNewSyncedPlaylists(playlists: List<PlaylistEntity>) {
    playlistDao.upsertPlaylists(playlists.map { it.copy(syncStatus = SyncStatus.SYNCED) })
}

suspend fun deleteLocalPlaylists(localIds: List<Long>) =
    localIds.forEach { playlistDao.deletePlaylist(it) }

suspend fun updateLocalPlaylistMetadata(localId: Long, remotePlaylist: PlaylistEntity) {
    // Use the surgical UPDATE query - this should NOT trigger CASCADE
    playlistDao.updatePlaylistMetadata(
        playlistId = localId,
        name = remotePlaylist.name,
        description = remotePlaylist.description,
        timestamp = remotePlaylist.last_modified_at
    )

    // Diagnostic: Verify items still exist after metadata update
    val itemsAfterUpdate = playlistDao.getItemsForPlaylist(localId).first()
```

```
    Timber.tag("SYNC_DEBUG").i(
        "After updateLocalPlaylistMetadata: Playlist $localId has ${itemsAfterUpdate.size}
    )
}


suspend fun getRemotePlaylistContent(serverId: String): List<MusicdexSong> =
    authenticatedMusicdexApiService.getPlaylistContent(serverId).body()?.content ?: emptyL

// --- MODIFIED TO FILTER LOCAL-ONLY ITEMS ---
suspend fun getLocalPlaylistItemServerIds(localPlaylistId: Long): List<String> =
    coroutineScope {
        val items = playlistDao.getItemsForPlaylist(localPlaylistId).first()

        // *** THE CRITICAL FILTER ***
        // Only consider items that are meant to be synced to the server.
        val syncedItems = items.filter { !it.isLocalOnly }

        // Asynchronously fetch the server ID for each item in the playlist.
        // This is necessary because the playlist only stores a reference (videoId + start
        // not the song's unique server UUID needed for the sync.
        syncedItems.map { playlistItem ->
            async {
                playlistItem.songStartSecondsPlaylist?.let { startTime ->
                    // This helper function fetches the video details from the API and fin
                    // song with the matching start time to get its server ID.
                    fetchVideoAndFindSong(playlistItem.videoIdForItem, startTime)?.second?
                }
            }
        }.awaitAll().filterNotNull() // Launch all lookups in parallel, wait for them, and
    }
// --- END OF MODIFICATION ---

suspend fun reconcileLocalPlaylistItems(localPlaylistId: Long, remoteSongs: List<MusicdexS
    appDatabase.withTransaction {
        // IMPORTANT: Read items at the START of the transaction to get fresh data
        // Using .first() on a Flow inside a transaction should give us the current state
        val localItems = playlistDao.getItemsForPlaylist(localPlaylistId).first()

        Timber.tag("SYNC_DEBUG").i(
            "reconcileLocalPlaylistItems START: Found ${localItems.size} local items for p
        )

        val localOnlyItems = localItems.filter { it.isLocalOnly }

        Timber.tag("SYNC_DEBUG").i(
            "reconcileLocalPlaylistItems: Filtered ${localOnlyItems.size} local-only items
        )

        // --- Step 2: Prepare the "Server Truth" ---
        val remoteItemEntities = remoteSongs.mapIndexedNotNull { index, song ->
            if (song.channelId.isNullOrBlank()) {
                Timber.w("Skipping playlist song ('${song.name}') because its top-level 'c
                return@mapIndexedNotNull null
            }
```

```
            PlaylistItemEntity(
                playlistOwnerId = localPlaylistId,
                itemIdInPlaylist = LikedItemEntity.generateSongItemId(
                    song.videoId,
                    song.start
                ),
                videoIdForItem = song.videoId,
                itemTypeInPlaylist = LikedItemType.SONG_SEGMENT,
                songStartSecondsPlaylist = song.start,
                songEndSecondsPlaylist = song.end,
                songNamePlaylist = song.name,
                songArtistTextPlaylist = song.channel.name,
                songArtworkUrlPlaylist = song.artUrl,
                itemOrder = index,
                syncStatus = SyncStatus.SYNCED,
                isLocalOnly = false
            )
        }

        // --- Step 3: Construct the new "Final Truth" by merging ---
        val finalMergedList = remoteItemEntities.toMutableList()
        finalMergedList.addAll(localOnlyItems)
        val finalListWithCorrectOrder = finalMergedList.mapIndexed { index, item ->
            item.copy(itemOrder = index)
        }

        // --- Step 4: Execute Database Operations ---
        // Delete all items first, then insert the merged list
        playlistDao.deleteAllItemsForPlaylist(localPlaylistId)

        if (finalListWithCorrectOrder.isNotEmpty()) {
            playlistDao.upsertPlaylistItems(finalListWithCorrectOrder)
        }

        Timber.tag("SYNC_DEBUG").i(
            "Reconciled playlist ID $localPlaylistId. Kept ${localOnlyItems.size} local-on
        )
    }
}

suspend fun savePlaylistEdits(
    editedPlaylist: PlaylistEntity,
    finalItems: List<PlaylistItemEntity>
) = withContext(defaultDispatcher) {
    val itemsToSave = finalItems.mapIndexed { index, item ->
        item.copy(itemOrder = index)
    }

    playlistDao.updatePlaylistAndItems(editedPlaylist, itemsToSave)

    if (editedPlaylist.syncStatus == SyncStatus.DIRTY) {
        Timber.tag(TAG_SYNC).i("Saved edits for playlist '${editedPlaylist.name}'. Marked
    } else {
        Timber.tag(TAG_SYNC).i("Saved local-only edits for playlist '${editedPlaylist.name
    }
```

```
}

suspend fun createNewPlaylist(name: String, description: String? = null): Long =
    withContext(defaultDispatcher) {
        val now = Instant.now().toString()
        val userId = tokenManager.getUserId()
        if (userId == null) {
            Timber.e("Cannot create playlist: User is not logged in.")
            throw IOException("User not logged in.")
        }
        playlistDao.insertPlaylist(
            PlaylistEntity(
                name = name.trim(),
                description = description?.trim(),
                syncStatus = SyncStatus.DIRTY,
                owner = userId.toLongOrNull(),
                createdAt = now,
                last_modified_at = now,
                isDeleted = false,
                serverId = null
            )
        )
    }

suspend fun deletePlaylist(playlistId: Long) = withContext(defaultDispatcher) {
    playlistDao.softDeletePlaylist(playlistId)
}

suspend fun addPlaylistItem(playlistItem: PlaylistItemEntity) = withContext(defaultDispatc
    appDatabase.withTransaction {
        // Only mark the playlist as dirty if the item being added is a syncable item.
        if (!playlistItem.isLocalOnly) {
            val parentPlaylist = playlistDao.getPlaylistById(playlistItem.playlistOwnerId)
            if (parentPlaylist != null) {
                playlistDao.updatePlaylist(
                    parentPlaylist.copy(
                        syncStatus = SyncStatus.DIRTY,
                        last_modified_at = Instant.now().toString()
                    )
                )
            }
        }

        // Always insert the new item, whether it's local or not.
        playlistDao.insertPlaylistItem(playlistItem)
    }
}
// Add these helper methods for diagnostics
suspend fun getPlaylistItemCount(localPlaylistId: Long): Int {
    return playlistDao.getItemsForPlaylist(localPlaylistId).first().size
}

suspend fun getLocalOnlyItemCount(localPlaylistId: Long): Int {
    return playlistDao.getItemsForPlaylist(localPlaylistId).first().count { it.isLocalOnly
}
```

```
// --- Favorite Channels ---
suspend fun performUpstreamFavoriteChannelsSync(logger: SyncLogger) {
    val pendingDeletes = favoriteChannelDao.getPendingDeletionIds()
    if (pendingDeletes.isNotEmpty()) {
        val patchRequest = pendingDeletes.map { PatchOperation(op = "remove", channel_id =
        val response = authenticatedMusicdexApiService.patchFavoriteChannels(patchRequest)
        if (response.isSuccessful) {
            logger.info("  -> Successfully sent ${pendingDeletes.size} favorite channel DE
            favoriteChannelDao.deleteSyncedDeletions(pendingDeletes)
        } else {
            logger.warning("  -> FAILED to send favorite channel deletions. Code: ${respon
        }
    }

    val dirtyItems = favoriteChannelDao.getDirtyItems()
    if (dirtyItems.isNotEmpty()) {
        val patchRequest = dirtyItems.map { PatchOperation(op = "add", channel_id = it.id)
        val response = authenticatedMusicdexApiService.patchFavoriteChannels(patchRequest)
        if (response.isSuccessful) {
            logger.info("  -> Successfully sent ${dirtyItems.size} favorite channel ADDITI
            favoriteChannelDao.markAsSynced(dirtyItems.map { it.id })
        } else {
            logger.warning("  -> FAILED to send favorite channel additions. Code: ${respon
        }
    }
}

suspend fun getRemoteFavoriteChannels(): List<FavoriteChannelEntity> {
    val response = holodexApiService.getFavoriteChannels()
    if (!response.isSuccessful) throw IOException("Failed to fetch remote favorite channel
    return response.body()?.map { it.toFavoriteChannelEntity() } ?: emptyList()
}

suspend fun getLocalFavoriteChannels(): List<FavoriteChannelEntity> =
    favoriteChannelDao.getFavoriteChannels().first()

suspend fun insertNewSyncedFavoriteChannels(channels: List<FavoriteChannelEntity>) =
    favoriteChannelDao.upsert(channels.map { it.copy(syncStatus = SyncStatus.SYNCED) })

suspend fun deleteLocalFavoriteChannels(channelIds: List<String>) =
    channelIds.forEach { favoriteChannelDao.softDelete(it) } // Using soft delete is safer

// --- Starred Playlists ---
suspend fun performUpstreamStarredPlaylistsSync(logger: SyncLogger) {
    val toRemove = starredPlaylistDao.getUnsyncedItems()
        .filter { it.syncStatus == SyncStatus.PENDING_DELETE }
    toRemove.forEach {
        val response =
            authenticatedMusicdexApiService.unstarPlaylist(StarPlaylistRequest(it.playlist
        if (response.isSuccessful) {
            logger.info("  -> Successfully UNSTARRED playlist ${it.playlistId} on server."
            starredPlaylistDao.deleteById(it.playlistId)
        } else {
            logger.warning("  -> FAILED to unstar playlist ${it.playlistId}. Code: ${respo
        }
```

```
    }

    val toAdd =
        starredPlaylistDao.getUnsyncedItems().filter { it.syncStatus == SyncStatus.DIRTY }
    toAdd.forEach {
        val response =
            authenticatedMusicdexApiService.starPlaylist(StarPlaylistRequest(it.playlistId
        if (response.isSuccessful) {
            logger.info("  -> Successfully STARRED playlist ${it.playlistId} on server.")
            starredPlaylistDao.insert(it.copy(syncStatus = SyncStatus.SYNCED))
        } else {
            logger.warning("  -> FAILED to star playlist ${it.playlistId}. Code: ${respons
        }
    }
}

suspend fun getRemoteStarredPlaylists(): List<PlaylistStub> =
    authenticatedMusicdexApiService.getStarredPlaylists().body() ?: emptyList()

suspend fun getLocalUnsyncedStarredPlaylistsCount(): Int =
    starredPlaylistDao.getUnsyncedItems().size

suspend fun deleteLocalSyncedStarredPlaylists(): Int {
    val syncedItems = starredPlaylistDao.getStarredPlaylists().first()
        .filter { it.syncStatus == SyncStatus.SYNCED }
    if (syncedItems.isNotEmpty()) {
        starredPlaylistDao.deleteAllSyncedItems() // Assuming this method deletes where sy
    }
    return syncedItems.size
}

suspend fun insertRemoteStarredPlaylistsAsSynced(starred: List<PlaylistStub>) {
    val entities = starred.map { StarredPlaylistEntity(it.id, SyncStatus.SYNCED) }
    starredPlaylistDao.upsertAll(entities)
}

private fun MusicdexSong.toHolodexSong(): HolodexSong {
    return HolodexSong(
        name = this.name,
        start = this.start,
        end = this.end,
        itunesId = null,
        artUrl = this.artUrl,
        originalArtist = this.originalArtist,
        videoId = this.videoId
    )
}



suspend fun searchMusicOnChannels(
    query: String,
    channelIds: List<String>
): Result<List<HolodexVideoItem>> = withContext(defaultDispatcher) {
    if (channelIds.isEmpty()) return@withContext Result.success(emptyList())
```

```
    try {
        val ytService = NewPipe.getService(ServiceList.YouTube.serviceId)
        val allResults = coroutineScope {
            channelIds.map { channelId ->
                async {
                    searchSingleChannel(ytService, channelId, query)
                }
            }.awaitAll()
        }

        Result.success(allResults.flatten())
    } catch (e: Exception) {
        Timber.e(e, "Failed to perform external channel search")
        Result.failure(e)
    }
}

private suspend fun searchSingleChannel(
    ytService: StreamingService,
    channelId: String,
    query: String
): List<HolodexVideoItem> = try {
    val channelUrl = "https://www.youtube.com/channel/$channelId"

    val channelExtractor = ytService.getChannelExtractor(channelUrl)
    channelExtractor.fetchPage()

    val avatarUrl = channelExtractor.avatars.firstOrNull()?.url.orEmpty()
    val channelName = channelExtractor.name

    val videosTab = channelExtractor.tabs.find { tab ->
        tab.contentFilters.contains("videos")
    }

    if (videosTab != null) {
        val tabExtractor = ytService.getChannelTabExtractor(videosTab)
        tabExtractor.fetchPage()

        tabExtractor.initialPage.items
            .mapNotNull { it as? StreamInfoItem }
            .filter { it.name.contains(query, ignoreCase = true) }
            .map { item ->
                mapStreamInfoItemToHolodexVideoItem(
                    item,
                    channelId,
                    channelName,
                    avatarUrl
                )
            }
            .filter { VideoFilteringUtil.isMusicContent(it) }
    } else {
        emptyList()
    }
} catch (e: Exception) {
    Timber.e(e, "Failed to search within channel $channelId")
```

```
    emptyList()
}



suspend fun getMusicFromExternalChannel(
    channelId: String,
    nextPage: org.schabi.newpipe.extractor.Page?
): Result<FetcherResult<HolodexVideoItem>> = withContext(defaultDispatcher) {
    try {
        val ytService = NewPipe.getService(ServiceList.YouTube.serviceId)
        val channelUrl = "https://www.youtube.com/channel/$channelId"
        val channelExtractor = ytService.getChannelExtractor(channelUrl)
        channelExtractor.fetchPage()

        val videosTab = channelExtractor.tabs.firstOrNull { it.getUrl().contains("/videos"
            ?: return@withContext Result.failure(Exception("Could not find Videos tab for

        val tabExtractor = ytService.getChannelTabExtractor(videosTab)
        val itemsPage = if (nextPage == null) {
            tabExtractor.fetchPage()
            tabExtractor.initialPage
        } else {
            tabExtractor.getPage(nextPage)
        }

        if (itemsPage == null || itemsPage.items.isEmpty()) {
            return@withContext Result.success(FetcherResult(emptyList(), null, null, null)
        }

        val videos = itemsPage.items.mapNotNull { it as? StreamInfoItem }
        val avatarUrl = channelExtractor.avatars.firstOrNull()?.url.orEmpty()

        val holodexItems = videos.map { item ->
            mapStreamInfoItemToHolodexVideoItem(
                item, channelId, channelExtractor.name, avatarUrl
            )
        }
        val musicContent = holodexItems.filter { VideoFilteringUtil.isMusicContent(it) }

        Result.success(
            FetcherResult(
                data = musicContent,
                totalAvailable = null,
                nextPageCursor = if (itemsPage.hasNextPage()) itemsPage.nextPage else null
            )
        )
    } catch (e: Exception) {
        Timber.e(e, "Failed to get music from external channel: $channelId")
        Result.failure(e)
    }
}


private fun mapStreamInfoItemToHolodexVideoItem(
```

```
    item: StreamInfoItem,
    channelId: String,
    channelName: String,
    channelAvatar: String
): HolodexVideoItem {
    val timestamp = try {
        item.uploadDate?.offsetDateTime()?.toInstant() ?: Instant.now()
    } catch (e: Exception) { Instant.now() }

    val videoId = try {
        item.url.substringAfter("watch?v=").substringBefore("&")
    } catch (e: Exception) { item.url }

    return HolodexVideoItem(
        id = videoId,
        title = item.name,
        type = "stream",
        topicId = null, // External items don't have Holodex topics
        availableAt = timestamp.toString(),
        publishedAt = timestamp.toString(),
        duration = item.duration.takeIf { it > 0 } ?: 0,
        status = "past", // Assume external videos are past
        channel = HolodexChannelMin(
            id = channelId,
            name = channelName,
            englishName = null,
            org = "External", // Mark as External
            type = "vtuber",
            photoUrl = channelAvatar
        ),
        songcount = 0, // External items don't have a song count from Holodex
        description = null, // StreamInfoItem doesn't have a full description, StreamExtra
        songs = null
    )
}

suspend fun searchForExternalChannels(query: String): Result<List<com.example.holodex.data
    try {
        val ytService = NewPipe.getService(ServiceList.YouTube.serviceId)
        val extractor = ytService.getSearchExtractor(query, listOf("channels"), "")
        extractor.fetchPage()

        val results = extractor.initialPage.items.mapNotNull { it as? org.schabi.newpipe.e
            .map { infoItem ->
                com.example.holodex.data.model.ChannelSearchResult(
                    channelId = infoItem.url.substringAfter("/channel/"),
                    name = infoItem.name,
                    thumbnailUrl = infoItem.thumbnails.firstOrNull()?.url,
                    subscriberCount = if (infoItem.subscriberCount > 0) "${infoItem.subscr
                )
            }
        Result.success(results)
    } catch (e: Exception) {
        Timber.e(e, "Failed to search for external channels with query: $query")
        Result.failure(e)
```

```
        }
    }

    suspend fun getOrganizationList(): Result<List<Organization>> = withContext(defaultDispatc
        try {
            val response = holodexApiService.getOrganizations()
            if (response.isSuccessful && response.body() != null) {
                Result.success(response.body()!!)
            } else {
                Result.failure(IOException("API Error fetching organizations: ${response.code(
            }
        } catch (e: Exception) {
            Result.failure(e)
        }
    }
}

// File: java\com\example\holodex\data\repository\LocalRepository.kt
package com.example.holodex.data.repository

import com.example.holodex.data.db.ExternalChannelEntity
import com.example.holodex.data.db.LocalDao
import com.example.holodex.data.db.LocalFavoriteEntity
import com.example.holodex.data.db.LocalPlaylistEntity
import com.example.holodex.data.db.LocalPlaylistItemEntity
import kotlinx.coroutines.flow.Flow
import javax.inject.Inject
import javax.inject.Singleton

/**
 * Repository dedicated to managing all local-only data.
 * This repository is NOT sync-aware and operates independently from HolodexRepository.
 */
@Singleton
class LocalRepository @Inject constructor(
    private val localDao: LocalDao
) {

    // --- Local Favorites ---
    suspend fun addLocalFavorite(favorite: LocalFavoriteEntity) = localDao.addLocalFavorite(fa
    suspend fun removeLocalFavorite(itemId: String) = localDao.removeLocalFavorite(itemId)
    fun getLocalFavorites(): Flow<List<LocalFavoriteEntity>> = localDao.getLocalFavorites()
    fun getLocalFavoriteIds(): Flow<List<String>> = localDao.getLocalFavoriteIds()

    // --- External Channels ---
    suspend fun addExternalChannel(channel: ExternalChannelEntity) = localDao.addExternalChann
    suspend fun removeExternalChannel(channelId: String) = localDao.removeExternalChannel(chan
    fun getAllExternalChannels(): Flow<List<ExternalChannelEntity>> = localDao.getAllExternalC
    suspend fun getExternalChannel(channelId: String): ExternalChannelEntity? = localDao.getEx

    // --- Local Playlists ---
    suspend fun createLocalPlaylist(playlist: LocalPlaylistEntity): Long = localDao.createLoca
    fun getAllLocalPlaylists(): Flow<List<LocalPlaylistEntity>> = localDao.getAllLocalPlaylist
    suspend fun addSongToLocalPlaylist(item: LocalPlaylistItemEntity) = localDao.addSongToLoca
    fun getItemsForLocalPlaylist(playlistId: Long): Flow<List<LocalPlaylistItemEntity>> = loca
```

```
}

// File: java\com\example\holodex\data\repository\SearchHistoryRepository.kt
// File: java/com/example/holodex/data/repository/SearchHistoryRepository.kt
package com.example.holodex.data.repository

import android.content.SharedPreferences
import androidx.core.content.edit
import com.google.gson.Gson
import com.google.gson.reflect.TypeToken
import kotlinx.coroutines.CoroutineDispatcher
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.withContext
import timber.log.Timber

interface SearchHistoryRepository {
    val searchHistory: StateFlow<List<String>>
    suspend fun addSearchQueryToHistory(query: String)
    suspend fun loadSearchHistory()
    suspend fun clearSearchHistory()
}

class SharedPreferencesSearchHistoryRepository(
    private val sharedPreferences: SharedPreferences,
    private val gson: Gson,
    private val ioDispatcher: CoroutineDispatcher = Dispatchers.IO
) : SearchHistoryRepository {

    private companion object {
        const val PREF_SEARCH_HISTORY = "search_history_list_v1" // Moved from ViewModel
        const val MAX_SEARCH_HISTORY_SIZE = 10 // Moved from ViewModel
        const val TAG = "SearchHistoryRepo"
    }

    private val _searchHistoryFlow = MutableStateFlow<List<String>>(emptyList())
    override val searchHistory: StateFlow<List<String>> = _searchHistoryFlow.asStateFlow()

    override suspend fun loadSearchHistory() {
        withContext(ioDispatcher) {
            try {
                val historyJson = sharedPreferences.getString(PREF_SEARCH_HISTORY, null)
                _searchHistoryFlow.value = if (historyJson != null) {
                    gson.fromJson(historyJson, object : TypeToken<List<String>>() {}.type) ?:
                } else {
                    emptyList()
                }
                Timber.tag(TAG).d("Search history loaded: ${_searchHistoryFlow.value.size} ite
            } catch (e: Exception) {
                Timber.tag(TAG).e(e, "Failed to load search history")
                _searchHistoryFlow.value = emptyList()
            }
        }
```

```
    }

    private suspend fun saveSearchHistory(newHistory: List<String>) {
        withContext(ioDispatcher) {
            try {
                sharedPreferences.edit { putString(PREF_SEARCH_HISTORY, gson.toJson(newHistory
                Timber.tag(TAG).d("Search history saved.")
            } catch (e: Exception) {
                Timber.tag(TAG).e(e, "Failed to save search history")
            }
        }
    }

    override suspend fun addSearchQueryToHistory(query: String) {
        val currentHistory = _searchHistoryFlow.value.toMutableList().apply {
            remove(query) // Remove if exists to move to top
            add(0, query) // Add to the beginning
        }
        _searchHistoryFlow.value = currentHistory.take(MAX_SEARCH_HISTORY_SIZE) // Trim to max
        saveSearchHistory(_searchHistoryFlow.value)
    }

    override suspend fun clearSearchHistory() {
        _searchHistoryFlow.value = emptyList()
        withContext(ioDispatcher) {
            sharedPreferences.edit { remove(PREF_SEARCH_HISTORY) }
        }
        Timber.tag(TAG).d("Search history cleared.")
    }
}

// File: java\com\example\holodex\data\repository\UserPreferencesRepository.kt
package com.example.holodex.data.repository

import android.content.Context
import androidx.datastore.core.DataStore
import androidx.datastore.preferences.core.Preferences
import androidx.datastore.preferences.core.booleanPreferencesKey
import androidx.datastore.preferences.core.edit
import androidx.datastore.preferences.preferencesDataStore
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.map
import timber.log.Timber

// Extension property to get the DataStore instance
val Context.userPreferencesDataStore: DataStore<Preferences> by preferencesDataStore(name = "u

class UserPreferencesRepository(private val dataStore: DataStore<Preferences>) {

    companion object {
        val AUTOPLAY_NEXT_VIDEO = booleanPreferencesKey("autoplay_next_video")
        val SHUFFLE_ON_PLAY_START = booleanPreferencesKey("shuffle_on_play_start")
        private const val TAG = "UserPrefsRepo"
    }
```

```
    val autoplayEnabled: Flow<Boolean> = dataStore.data
        .map { preferences ->
            preferences[AUTOPLAY_NEXT_VIDEO] ?: true // Default to true (autoplay is on)
        }
        .also { Timber.d("$TAG: Observing autoplayEnabled preference.") }

    val shuffleOnPlayStartEnabled: Flow<Boolean> = dataStore.data
        .map { preferences ->
            preferences[SHUFFLE_ON_PLAY_START] ?: false // Default to false (OFF)
        }

    suspend fun setAutoplayEnabled(enabled: Boolean) {
        dataStore.edit { preferences ->
            preferences[AUTOPLAY_NEXT_VIDEO] = enabled
        }
        Timber.i("$TAG: Autoplay preference set to: $enabled")
    }
    suspend fun setShuffleOnPlayStartEnabled(enabled: Boolean) {
        dataStore.edit { preferences ->
            preferences[SHUFFLE_ON_PLAY_START] = enabled
        }
        Timber.i("$TAG: Shuffle on play start preference set to: $enabled")
    }
}


// File: java\com\example\holodex\data\repository\YouTubeStreamRepository.kt
// File: java\com\example\holodex\data\repository\YouTubeStreamRepository.kt
package com.example.holodex.data.repository

import android.content.SharedPreferences
import com.example.holodex.data.model.AudioStreamDetails
import com.example.holodex.viewmodel.AppPreferenceConstants
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.withContext
import org.schabi.newpipe.extractor.NewPipe
import org.schabi.newpipe.extractor.ServiceList
import org.schabi.newpipe.extractor.stream.AudioStream
import org.schabi.newpipe.extractor.stream.AudioTrackType
import org.schabi.newpipe.extractor.stream.StreamInfo
import timber.log.Timber
import java.util.Locale
import javax.inject.Inject
import javax.inject.Singleton

@Singleton
class YouTubeStreamRepository @Inject constructor(
    private val sharedPreferences: SharedPreferences,
) {

    companion object {
        private const val TAG = "YouTubeStreamRepo"
    }

    private val saverMaxBitrate = 96
```

```
private val standardMaxBitrate = 140

suspend fun getAudioStreamDetails(videoId: String): Result<AudioStreamDetails> {
    return withContext(Dispatchers.IO) {
        try {
            val youtubeUrl = "https://www.youtube.com/watch?v=$videoId"
            val ytService = NewPipe.getService(ServiceList.YouTube.serviceId)
                ?: return@withContext Result.failure(Exception("YouTube service not found.

            val streamInfo: StreamInfo = StreamInfo.getInfo(ytService, youtubeUrl)

            val allAudioStreams: List<AudioStream> = streamInfo.audioStreams

            if (allAudioStreams.isEmpty()) {
                return@withContext Result.failure(Exception("No audio streams found for vi
            }

            val audioQualityPref = sharedPreferences.getString(
                AppPreferenceConstants.PREF_AUDIO_QUALITY,
                AppPreferenceConstants.AUDIO_QUALITY_BEST
            ) ?: AppPreferenceConstants.AUDIO_QUALITY_BEST

            val applyQualityFilterAndSort: (List<AudioStream>) -> AudioStream? = { streams
                val qualityFiltered = when (audioQualityPref) {
                    AppPreferenceConstants.AUDIO_QUALITY_SAVER ->
                        streams.filter { it.averageBitrate in 1..saverMaxBitrate }
                            .ifEmpty { streams.filter { it.averageBitrate in 1..standardMa
                            .ifEmpty { streams }
                    AppPreferenceConstants.AUDIO_QUALITY_STANDARD ->
                        streams.filter { it.averageBitrate in 1..standardMaxBitrate }
                            .ifEmpty { streams }
                    else -> streams
                }
                qualityFiltered.maxByOrNull { it.averageBitrate }
            }

            // --- FIX: Expanded fallback to include OPUS and WEBM with priority ---
            val bestAudioStream =
                // Priority 1: Original Japanese track
                applyQualityFilterAndSort(allAudioStreams.filter {
                    it.audioTrackType == AudioTrackType.ORIGINAL && it.audioLocale?.langua
                })
                // Priority 2: Any Original track
                    ?: applyQualityFilterAndSort(allAudioStreams.filter {
                        it.audioTrackType == AudioTrackType.ORIGINAL
                    })
                    // Priority 3: Any Japanese track
                    ?: applyQualityFilterAndSort(allAudioStreams.filter {
                        it.audioLocale?.language == Locale.JAPANESE.language
                    })
                    // Final Fallback: The best of whatever is left
                    ?: applyQualityFilterAndSort(allAudioStreams)


            if (bestAudioStream != null) {
```

```
                    val finalUrl = bestAudioStream.content
                        ?: return@withContext Result.failure(Exception("Selected best audio st
                    val finalFormat = bestAudioStream.format?.getName()?.uppercase() ?: "UNKNO
                    val qualityDesc = "${bestAudioStream.averageBitrate}kbps"
                    val trackType = bestAudioStream.audioTrackType?.name ?: "UNKNOWN_TYPE"
                    Timber.i("$TAG: Resolved stream for $videoId. Quality: '$audioQualityPref'
                    return@withContext Result.success(
                        AudioStreamDetails(
                            streamUrl = finalUrl,
                            format = finalFormat,
                            quality = qualityDesc
                        )
                    )
                } else {
                    Timber.e("$TAG: Could not select a best audio stream for $videoId after al
                    return@withContext Result.failure(Exception("No suitable audio stream foun
                }

            } catch (e: Exception) {
                Timber.e(e, "$TAG: Unexpected error for $videoId")
                return@withContext Result.failure(
                    Exception("An unexpected error occurred while fetching stream for $videoId
                )
            }
        }
    }
}

// File: java\com\example\holodex\data\source\CacheSchemeDataSourceFactory.kt
// File: java/com/example/holodex/data/source/CacheSchemeDataSourceFactory.kt

package com.example.holodex.data.source

import androidx.media3.common.util.UnstableApi
import androidx.media3.datasource.DataSource
import androidx.media3.datasource.DataSpec // <-- ADD THIS IMPORT
import androidx.media3.datasource.DefaultHttpDataSource
import androidx.media3.datasource.cache.CacheDataSource
import androidx.media3.datasource.cache.SimpleCache
import timber.log.Timber

/**
 * A custom DataSource.Factory that routes requests based on the URI scheme.
 * - Handles "http" and "https" by delegating to a standard HttpDataSource.
 * - Handles a custom "cache" scheme by delegating to a CacheDataSource,
 *   using the URI's authority as the cache key.
 */
@UnstableApi
class CacheSchemeDataSourceFactory(
    private val downloadCache: SimpleCache,
    private val upstreamFactory: DefaultHttpDataSource.Factory
) : DataSource.Factory {

    override fun createDataSource(): DataSource {
        val defaultCacheDataSource = CacheDataSource(
```

```
            downloadCache,
            upstreamFactory.createDataSource(),
            CacheDataSource.FLAG_BLOCK_ON_CACHE or CacheDataSource.FLAG_IGNORE_CACHE_ON_ERROR
        )

        val cacheOnlyDataSource = CacheDataSource(downloadCache, null)

        return object : DataSource by defaultCacheDataSource {
            // The method signature is now corrected to match the DataSource interface.
            // It takes a single `DataSpec` parameter.
            override fun open(dataSpec: DataSpec): Long {
                return if (dataSpec.uri.scheme == "cache") {
                    Timber.d("CacheScheme: Routing 'cache://' URI to cache-only source. Key: $
                    // The logic inside remains correct. We create a new DataSpec using the au
                    val newSpec = dataSpec.buildUpon().setKey(dataSpec.uri.authority).build()
                    cacheOnlyDataSource.open(newSpec)
                } else {
                    // For all other schemes (http, https), use the default CacheDataSource.
                    Timber.d("CacheScheme: Routing '${dataSpec.uri.scheme}' URI to default cac
                    defaultCacheDataSource.open(dataSpec)
                }
            }
        }
    }
}

// File: java\com\example\holodex\di\AppModule.kt
// File: java\com\example\holodex\di\AppModule.kt
package com.example.holodex.di

import android.app.Application
import android.content.Context
import android.content.SharedPreferences
import androidx.annotation.OptIn
import androidx.media3.exoplayer.offline.DownloadNotificationHelper
import androidx.work.WorkManager
import coil.ImageLoader
import com.example.holodex.playback.PlaybackRequestManager
import com.example.holodex.util.PaletteExtractor
import com.google.gson.Gson
import dagger.Module
import dagger.Provides
import dagger.hilt.InstallIn
import dagger.hilt.android.UnstableApi
import dagger.hilt.android.qualifiers.ApplicationContext
import dagger.hilt.components.SingletonComponent
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.SupervisorJob
import javax.inject.Singleton

@Module
@InstallIn(SingletonComponent::class)
object AppModule {
```

```
@Provides
@Singleton
fun provideWorkManager(@ApplicationContext context: Context): WorkManager {
    return WorkManager.getInstance(context)
}

@Provides
@Singleton
fun provideSharedPreferences(app: Application): SharedPreferences {
    return app.getSharedPreferences("UserPrefs", Context.MODE_PRIVATE)
}

@Provides
@Singleton
fun provideGson(): Gson = Gson()

@Provides
@Singleton
fun providePlaybackRequestManager(): PlaybackRequestManager = PlaybackRequestManager()

@Provides
@Singleton
@ApplicationScope // We'll create this annotation for clarity
fun provideApplicationScope(): CoroutineScope =
    CoroutineScope(SupervisorJob() + Dispatchers.Default)

@Provides
@Singleton
fun provideImageLoader(@ApplicationContext context: Context): ImageLoader {
    // This logic is moved from MyApp.kt's newImageLoader()
    return ImageLoader.Builder(context)
        .memoryCache {
            coil.memory.MemoryCache.Builder(context)
                .maxSizePercent(0.25)
                .build()
        }
        .diskCache {
            coil.disk.DiskCache.Builder()
                .directory(context.cacheDir.resolve("image_cache_v1"))
                .maxSizeBytes(50L * 1024L * 1024L) // 50MB
                .build()
        }
        .respectCacheHeaders(false)
        .build()
}

@OptIn(androidx.media3.common.util.UnstableApi::class)
@Provides
@Singleton
@UnstableApi
fun provideDownloadNotificationHelper(@ApplicationContext context: Context): DownloadNotif
    return DownloadNotificationHelper(context, "download_channel")
}
@Provides
@Singleton
```

```
    fun providePaletteExtractor(@ApplicationContext context: Context): PaletteExtractor {
        return PaletteExtractor(context)
    }
}

// File: java\com\example\holodex\di\AuthModule.kt
package com.example.holodex.di

import android.content.Context
import com.example.holodex.auth.TokenManager
import dagger.Module
import dagger.Provides
import dagger.hilt.InstallIn
import dagger.hilt.android.qualifiers.ApplicationContext
import dagger.hilt.components.SingletonComponent
import net.openid.appauth.AuthorizationService
import javax.inject.Singleton

@Module
@InstallIn(SingletonComponent::class)
object AuthModule {

    @Provides
    @Singleton
    fun provideTokenManager(@ApplicationContext context: Context): TokenManager {
        return TokenManager(context)
    }

    @Provides
    // Not a singleton, as it should be created and disposed with the component that uses it (
    fun provideAuthorizationService(@ApplicationContext context: Context): AuthorizationServic
        return AuthorizationService(context)
    }
}

// File: java\com\example\holodex\di\CacheModule.kt
package com.example.holodex.di

import android.content.Context
import androidx.media3.common.util.UnstableApi
import androidx.media3.database.StandaloneDatabaseProvider
import androidx.media3.datasource.cache.LeastRecentlyUsedCacheEvictor
import androidx.media3.datasource.cache.NoOpCacheEvictor
import androidx.media3.datasource.cache.SimpleCache
import com.example.holodex.data.cache.BrowseListCache
import com.example.holodex.data.cache.SearchListCache
import com.example.holodex.data.db.BrowsePageDao
import com.example.holodex.data.db.SearchPageDao
import dagger.Module
import dagger.Provides
import dagger.hilt.InstallIn
import dagger.hilt.android.qualifiers.ApplicationContext
import dagger.hilt.components.SingletonComponent
import java.io.File
import javax.inject.Singleton
```

```
@Module
@InstallIn(SingletonComponent::class)
object CacheModule {

    @Provides
    @Singleton
    fun provideBrowseListCache(browsePageDao: BrowsePageDao): BrowseListCache {
        return BrowseListCache(browsePageDao)
    }

    @Provides
    @Singleton
    fun provideSearchListCache(searchPageDao: SearchPageDao): SearchListCache {
        return SearchListCache(searchPageDao)
    }

    @Provides
    @Singleton
    @DownloadCache
    @UnstableApi
    fun provideDownloadCache(@ApplicationContext context: Context): SimpleCache {
        val downloadDirectory = File(context.getExternalFilesDir(null), "downloads")
        val databaseProvider = StandaloneDatabaseProvider(context)
        return SimpleCache(downloadDirectory, NoOpCacheEvictor(), databaseProvider)
    }

    @Provides
    @Singleton
    @MediaCache
    @UnstableApi
    fun provideMediaCache(@ApplicationContext context: Context): SimpleCache {
        val mediaCacheDirectory = File(context.cacheDir, "exoplayer_media_cache")
        val cacheEvictor = LeastRecentlyUsedCacheEvictor(150L * 1024L * 1024L) // 150MB
        val databaseProvider = StandaloneDatabaseProvider(context)
        return SimpleCache(mediaCacheDirectory, cacheEvictor, databaseProvider)
    }
}

// File: java\com\example\holodex\di\DatabaseModule.kt
// File: java/com/example/holodex/di/DatabaseModule.kt
package com.example.holodex.di

import android.app.Application
import com.example.holodex.data.db.AppDatabase
import com.example.holodex.data.db.BrowsePageDao
import com.example.holodex.data.db.DiscoveryDao
import com.example.holodex.data.db.DownloadedItemDao
import com.example.holodex.data.db.FavoriteChannelDao
import com.example.holodex.data.db.HistoryDao
import com.example.holodex.data.db.LikedItemDao
import com.example.holodex.data.db.LocalDao
import com.example.holodex.data.db.ParentVideoMetadataDao
import com.example.holodex.data.db.PlaylistDao
import com.example.holodex.data.db.SearchPageDao
```

```
import com.example.holodex.data.db.StarredPlaylistDao
import com.example.holodex.data.db.SyncMetadataDao
import com.example.holodex.data.db.VideoDao
import com.example.holodex.playback.data.model.PersistedPlaybackStateDao
import dagger.Module
import dagger.Provides
import dagger.hilt.InstallIn
import dagger.hilt.components.SingletonComponent
import javax.inject.Singleton

@Module
@InstallIn(SingletonComponent::class)
object DatabaseModule {

    @Provides
    @Singleton
    fun provideAppDatabase(app: Application): AppDatabase = AppDatabase.getDatabase(app)

    @Provides
    fun provideVideoDao(db: AppDatabase): VideoDao = db.videoDao()

    @Provides
    fun provideLikedItemDao(db: AppDatabase): LikedItemDao = db.likedItemDao()

    @Provides
    fun providePlaylistDao(db: AppDatabase): PlaylistDao = db.playlistDao()

    @Provides
    fun provideBrowsePageDao(db: AppDatabase): BrowsePageDao = db.browsePageDao()

    @Provides
    fun provideSearchPageDao(db: AppDatabase): SearchPageDao = db.searchPageDao()

    @Provides
    fun provideDownloadedItemDao(db: AppDatabase): DownloadedItemDao = db.downloadedItemDao()

    @Provides
    fun provideHistoryDao(db: AppDatabase): HistoryDao = db.historyDao()

    @Provides
    fun provideFavoriteChannelDao(db: AppDatabase): FavoriteChannelDao = db.favoriteChannelDao

    @Provides
    fun provideDiscoveryDao(db: AppDatabase): DiscoveryDao = db.discoveryDao()

    @Provides
    fun provideLocalDao(db: AppDatabase): LocalDao = db.localDao()

    @Provides
    fun provideParentVideoMetadataDao(db: AppDatabase): ParentVideoMetadataDao =
        db.parentVideoMetadataDao()

    @Provides
    fun providePersistedPlaybackStateDao(db: AppDatabase): PersistedPlaybackStateDao =
        db.persistedPlaybackStateDao()
```

```
    @Provides
    fun provideSyncMetadataDao(db: AppDatabase): SyncMetadataDao = db.syncMetadataDao()

    @Provides
    fun provideStarredPlaylistDao(db: AppDatabase): StarredPlaylistDao = db.starredPlaylistDao
}

// File: java\com\example\holodex\di\NetworkModule.kt
// File: java/com/example/holodex/di/NetworkModule.kt

package com.example.holodex.di

import android.content.SharedPreferences
import com.example.holodex.auth.TokenManager
import com.example.holodex.data.api.AuthenticatedMusicdexApiService
import com.example.holodex.data.api.HolodexApiService
import com.example.holodex.data.api.MusicdexApiService
import com.google.gson.Gson
import dagger.Module
import dagger.Provides
import dagger.hilt.InstallIn
import dagger.hilt.components.SingletonComponent
import okhttp3.OkHttpClient
import okhttp3.logging.HttpLoggingInterceptor
import retrofit2.Retrofit
import retrofit2.converter.gson.GsonConverterFactory
import timber.log.Timber
import java.util.concurrent.TimeUnit
import javax.inject.Singleton

@Module
@InstallIn(SingletonComponent::class)
object NetworkModule {

    private const val BROWSER_USER_AGENT = "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWeb

    @Provides
    @Singleton
    fun provideLoggingInterceptor(): HttpLoggingInterceptor {
        return HttpLoggingInterceptor { message ->
            Timber.tag("OkHttp-Holodex").i(message)
        }.apply { level = HttpLoggingInterceptor.Level.BODY }
    }

    // CLIENT FOR HOLODEX.NET API (GET requests, etc.)
    @Provides
    @Singleton
    @HolodexHttpClient
    fun provideHolodexOkHttpClient(
        sharedPreferences: SharedPreferences,
        tokenManager: TokenManager,
        loggingInterceptor: HttpLoggingInterceptor
    ): OkHttpClient {
        return OkHttpClient.Builder()
```

```
        .addInterceptor { chain ->
            val requestBuilder = chain.request().newBuilder()
            requestBuilder.header("User-Agent", BROWSER_USER_AGENT)
            val apiKey = sharedPreferences.getString("API_KEY", "") ?: ""
            if (apiKey.isNotEmpty()) {
                requestBuilder.header("X-APIKEY", apiKey)
            }
            // Also add JWT if available, for endpoints like GET /users/favorites
            tokenManager.getJwt()?.let { jwt ->
                requestBuilder.header("Authorization", "Bearer $jwt")
            }
            chain.proceed(requestBuilder.build())
        }
        .addInterceptor(loggingInterceptor)
        .connectTimeout(90, TimeUnit.SECONDS)
        .readTimeout(90, TimeUnit.SECONDS)
        .build()
}

// CLIENT FOR PUBLIC MUSICDEX.NET API (API Key only)
@Provides
@Singleton
@MusicdexHttpClient
fun provideMusicdexOkHttpClient(
    sharedPreferences: SharedPreferences,
    loggingInterceptor: HttpLoggingInterceptor
): OkHttpClient {
    return OkHttpClient.Builder()
        .addInterceptor { chain ->
            val requestBuilder = chain.request().newBuilder()
            requestBuilder.header("User-Agent", BROWSER_USER_AGENT)
            val apiKey = sharedPreferences.getString("API_KEY", "") ?: ""
            if (apiKey.isNotEmpty()) {
                requestBuilder.header("X-APIKEY", apiKey)
            }
            chain.proceed(requestBuilder.build())
        }
        .addInterceptor(loggingInterceptor)
        .connectTimeout(90, TimeUnit.SECONDS)
        .readTimeout(90, TimeUnit.SECONDS)
        .build()
}

// CLIENT FOR AUTHENTICATED MUSICDEX.NET API (Likes, History, Playlists, AND NOW FAVORITES
// --- FIX: This client now sends both the API Key and the JWT ---
// This makes it compatible with the favorites PATCH endpoint without breaking the likes e
@Provides
@Singleton
@AuthenticatedMusicdexHttpClient
fun provideAuthenticatedMusicdexOkHttpClient(
    sharedPreferences: SharedPreferences, // <-- ADDED
    tokenManager: TokenManager,
    loggingInterceptor: HttpLoggingInterceptor
): OkHttpClient {
    return OkHttpClient.Builder()
```

```
        .addInterceptor { chain ->
            val requestBuilder = chain.request().newBuilder()
            requestBuilder.header("User-Agent", BROWSER_USER_AGENT)

            // Add API Key
            val apiKey = sharedPreferences.getString("API_KEY", "") ?: ""
            if (apiKey.isNotEmpty()) {
                requestBuilder.header("X-APIKEY", apiKey)
            }

            // Add JWT
            tokenManager.getJwt()?.let { jwt ->
                requestBuilder.header("Authorization", "Bearer $jwt")
            }
            requestBuilder.header("Referer", "https://music.holodex.net/")

            chain.proceed(requestBuilder.build())
        }
        .addInterceptor(loggingInterceptor)
        .connectTimeout(90, TimeUnit.SECONDS)
        .readTimeout(90, TimeUnit.SECONDS)
        .build()
}

// --- RETROFIT SERVICE PROVIDERS (unchanged) ---

@Provides
@Singleton
fun provideHolodexApiService(@HolodexHttpClient okHttpClient: OkHttpClient): HolodexApiSer
    return Retrofit.Builder()
        .baseUrl("https://holodex.net/")
        .client(okHttpClient)
        .addConverterFactory(GsonConverterFactory.create())
        .build()
        .create(HolodexApiService::class.java)
}

@Provides
@Singleton
fun provideMusicdexApiService(@MusicdexHttpClient okHttpClient: OkHttpClient, gson: Gson):
    return Retrofit.Builder()
        .baseUrl("https://music.holodex.net/")
        .client(okHttpClient)
        .addConverterFactory(GsonConverterFactory.create(gson))
        .build()
        .create(MusicdexApiService::class.java)
}

@Provides
@Singleton
fun provideAuthenticatedMusicdexApiService(@AuthenticatedMusicdexHttpClient okHttpClient:
    return Retrofit.Builder()
        .baseUrl("https://music.holodex.net/")
        .client(okHttpClient)
        .addConverterFactory(GsonConverterFactory.create(gson))
```

```
            .build()
            .create(AuthenticatedMusicdexApiService::class.java)
    }
}

// File: java\com\example\holodex\di\PlaybackModule.kt
// File: java/com/example/holodex/di/PlaybackModule.kt
package com.example.holodex.di

import android.content.Context
import android.content.SharedPreferences
import androidx.media3.common.C
import androidx.media3.common.Player
import androidx.media3.common.util.UnstableApi
import androidx.media3.database.StandaloneDatabaseProvider
import androidx.media3.datasource.DataSource
import androidx.media3.datasource.DataSpec
import androidx.media3.datasource.DefaultDataSource
import androidx.media3.datasource.DefaultHttpDataSource
import androidx.media3.datasource.cache.CacheDataSource
import androidx.media3.datasource.cache.SimpleCache
import androidx.media3.exoplayer.DefaultLoadControl
import androidx.media3.exoplayer.ExoPlayer
import androidx.media3.exoplayer.offline.DownloadManager
import androidx.media3.exoplayer.source.DefaultMediaSourceFactory
import androidx.media3.exoplayer.source.preload.DefaultPreloadManager
import androidx.media3.exoplayer.upstream.DefaultAllocator
import com.example.holodex.data.db.DownloadedItemDao
import com.example.holodex.data.repository.DownloadRepository
import com.example.holodex.data.repository.HolodexRepository
import com.example.holodex.data.repository.UserPreferencesRepository
import com.example.holodex.playback.data.mapper.MediaItemMapper
import com.example.holodex.playback.data.model.PersistedPlaybackStateDao
import com.example.holodex.playback.data.persistence.PlaybackStatePersistenceManager
import com.example.holodex.playback.data.preload.PreloadConfiguration
import com.example.holodex.playback.data.preload.PreloadStatusController
import com.example.holodex.playback.data.queue.PlaybackQueueManager
import com.example.holodex.playback.data.queue.ShuffleOrderProvider
import com.example.holodex.playback.data.repository.HolodexStreamResolverRepositoryImpl
import com.example.holodex.playback.data.repository.Media3PlaybackRepositoryImpl
import com.example.holodex.playback.data.repository.RoomPlaybackStateRepositoryImpl
import com.example.holodex.playback.data.source.StreamResolutionCoordinator
import com.example.holodex.playback.data.tracker.PlaybackProgressTracker
import com.example.holodex.playback.domain.repository.PlaybackRepository
import com.example.holodex.playback.domain.repository.PlaybackStateRepository
import com.example.holodex.playback.domain.repository.StreamResolverRepository
import com.example.holodex.playback.player.Media3PlayerController
import com.example.holodex.viewmodel.AppPreferenceConstants
import com.example.holodex.viewmodel.autoplay.AutoplayItemProvider
import com.example.holodex.viewmodel.autoplay.ContinuationManager
import dagger.Module
import dagger.Provides
import dagger.hilt.InstallIn
import dagger.hilt.android.qualifiers.ApplicationContext
import dagger.hilt.components.SingletonComponent
```

```
import kotlinx.coroutines.CoroutineScope
import timber.log.Timber
import java.util.concurrent.Executors
import javax.inject.Singleton

private data class BufferSettings(
    val minBufferMs: Int,
    val maxBufferMs: Int,
    val bufferForPlaybackMs: Int,
    val bufferForPlaybackAfterRebufferMs: Int
)

@Module
@InstallIn(SingletonComponent::class)
@UnstableApi
object PlaybackModule {

    @Provides
    @Singleton
    fun provideLoadControl(sharedPreferences: SharedPreferences): DefaultLoadControl {
        val bufferingStrategy = sharedPreferences.getString(
            AppPreferenceConstants.PREF_BUFFERING_STRATEGY,
            AppPreferenceConstants.BUFFERING_STRATEGY_AGGRESSIVE
        ) ?: AppPreferenceConstants.BUFFERING_STRATEGY_AGGRESSIVE

        val settings = when (bufferingStrategy) {
            AppPreferenceConstants.BUFFERING_STRATEGY_BALANCED -> {
                Timber.d("ExoPlayer LoadControl: BALANCED")
                BufferSettings(20000, 60000, 3000, 5000)
            }
            AppPreferenceConstants.BUFFERING_STRATEGY_STABLE -> {
                Timber.d("ExoPlayer LoadControl: STABLE")
                BufferSettings(30000, 120000, 7500, 10000)
            }
            else -> {
                Timber.d("ExoPlayer LoadControl: AGGRESSIVE (default)");
                BufferSettings(10000, 60000, 1000, 2500)
            }
        }

        return DefaultLoadControl.Builder()
            .setAllocator(DefaultAllocator(true, C.DEFAULT_BUFFER_SEGMENT_SIZE))
            .setBufferDurationsMs(
                settings.minBufferMs,
                settings.maxBufferMs,
                settings.bufferForPlaybackMs,
                settings.bufferForPlaybackAfterRebufferMs
            )
            .build()
    }

    @Provides
    @Singleton
    fun provideExoPlayer(@ApplicationContext context: Context, loadControl: DefaultLoadControl
        return ExoPlayer.Builder(context)
```

```
        .setMediaSourceFactory(mediaSourceFactory)
        .setLoadControl(loadControl)
        .build()
}

@Provides
@Singleton
fun providePlayer(exoPlayer: ExoPlayer): Player = exoPlayer

@Provides
@Singleton
fun provideDownloadManager(
    @ApplicationContext context: Context,
    @DownloadCache downloadCache: SimpleCache
): DownloadManager {
    val databaseProvider = StandaloneDatabaseProvider(context)
    val downloadManagerDataSourceFactory = DefaultHttpDataSource.Factory().setUserAgent("H
    return DownloadManager(
        context,
        databaseProvider,
        downloadCache,
        downloadManagerDataSourceFactory,
        Executors.newFixedThreadPool(3)
    ).apply { resumeDownloads() }
}

@Provides
@Singleton
fun provideDataSourceFactory(
    @ApplicationContext context: Context,
    @DownloadCache downloadCache: SimpleCache,
    @MediaCache mediaCache: SimpleCache
): DataSource.Factory {
    // --- START OF REPLACEMENT ---

    // 1. The base factory for network requests.
    val upstreamFactory = DefaultHttpDataSource.Factory().setUserAgent("HolodexApp/1.0")

    // 2. The default factory that handles most schemes (http, https, content, file, etc.)
    val defaultDataSourceFactory = DefaultDataSource.Factory(context, upstreamFactory)

    // 3. The factory that handles streaming from the media cache.
    val mediaCacheDataSourceFactory = CacheDataSource.Factory()
        .setCache(mediaCache)
        .setUpstreamDataSourceFactory(defaultDataSourceFactory) // Use the default factory

    // 4. Our final, all-encompassing factory.
    return object : DataSource.Factory {
        override fun createDataSource(): DataSource {
            // The source for handling downloads via the "cache://" scheme.
            val downloadCacheDataSource = CacheDataSource(downloadCache, null) // Cache-on

            // The source for everything else (network streaming, local files).
            val defaultSource = mediaCacheDataSourceFactory.createDataSource()
```

```
            return object : DataSource by defaultSource {
                override fun open(dataSpec: DataSpec): Long {
                    // --- FIX: Add a specific case for the 'placeholder' scheme ---
                    return when (dataSpec.uri.scheme) {
                        "cache" -> {
                            // If it's our special download scheme, use the download cache
                            Timber.d("DataSource: Routing 'cache://' to download cache. Ke
                            val newSpec = dataSpec.buildUpon().setKey(dataSpec.uri.authori
                            downloadCacheDataSource.open(newSpec)
                        }
                        "placeholder" -> {
                            // If it's a placeholder, don't open anything. Return 0 bytes.
                            // This prevents the HttpDataSource from ever seeing it.
                            Timber.d("DataSource: Intercepting 'placeholder://' URI. Retur
                            0
                        }
                        else -> {
                            // For everything else (http, https, content), use the default
                            Timber.d("DataSource: Routing '${dataSpec.uri.scheme}' to defa
                            defaultSource.open(dataSpec)
                        }
                    }
                    // --- END OF FIX ---
                }
            }
        }
    }
    // --- END OF REPLACEMENT ---
}


@Provides
@Singleton
fun provideDefaultMediaSourceFactory(@ApplicationContext context: Context, dataSourceFacto
    return DefaultMediaSourceFactory(context).setDataSourceFactory(dataSourceFactory)
}

@Provides
@Singleton
fun providePreloadManager(
    @ApplicationContext context: Context,
    statusController: PreloadStatusController,
    mediaSourceFactory: DefaultMediaSourceFactory,
    loadControl: DefaultLoadControl
): DefaultPreloadManager {
    return DefaultPreloadManager.Builder(context, statusController)
        .setMediaSourceFactory(mediaSourceFactory)
        .setLoadControl(loadControl)
        .build()
}

@Provides
@Singleton
fun provideMedia3PlayerController(
    @ApplicationContext context: Context,
```

```
    exoPlayer: ExoPlayer,
    preloadManager: DefaultPreloadManager
): Media3PlayerController {
    return Media3PlayerController(exoPlayer, preloadManager)
}

@Provides
@Singleton
fun providePlaybackStateRepository(dao: PersistedPlaybackStateDao): PlaybackStateRepositor
    return RoomPlaybackStateRepositoryImpl(dao)
}

@Provides
@Singleton
fun provideStreamResolverRepository(repo: com.example.holodex.data.repository.YouTubeStrea
    return HolodexStreamResolverRepositoryImpl(repo)
}

@Provides
@Singleton
fun providePlaybackRepository(
    playerController: Media3PlayerController,
    queueManager: PlaybackQueueManager,
    streamResolver: StreamResolutionCoordinator,
    progressTracker: PlaybackProgressTracker,
    persistenceManager: PlaybackStatePersistenceManager,
    continuationManager: ContinuationManager,
    mediaItemMapper: MediaItemMapper,
    downloadRepository: DownloadRepository,
    holodexRepository: HolodexRepository,
    userPreferencesRepository: UserPreferencesRepository,
    @ApplicationScope scope: CoroutineScope
): PlaybackRepository {
    return Media3PlaybackRepositoryImpl(
        playerController,
        queueManager,
        streamResolver,
        progressTracker,
        persistenceManager,
        continuationManager,
        mediaItemMapper,
        downloadRepository,
        holodexRepository,
        userPreferencesRepository,
        scope
    )
}

@Provides
@Singleton
fun provideMediaItemMapper(@ApplicationContext context: Context, sharedPreferences: Shared

@Provides
@Singleton
fun provideStreamResolutionCoordinator(repo: StreamResolverRepository, dao: DownloadedItem
```

```
@Provides
@Singleton
fun provideShuffleOrderProvider(): ShuffleOrderProvider = ShuffleOrderProvider()

@Provides
@Singleton
fun providePlaybackQueueManager(provider: ShuffleOrderProvider): PlaybackQueueManager =
    PlaybackQueueManager(provider)

@Provides
@Singleton
fun providePlaybackProgressTracker(
    player: Player,
    @ApplicationScope scope: CoroutineScope,
    holodexRepository: HolodexRepository,
    mediaItemMapper: MediaItemMapper
): PlaybackProgressTracker {
    return PlaybackProgressTracker(
        player,
        scope,
        holodexRepository,
        mediaItemMapper
    )
}

@Provides
@Singleton
fun providePlaybackStatePersistenceManager(
    repo: PlaybackStateRepository,
    @ApplicationScope scope: CoroutineScope
): PlaybackStatePersistenceManager = PlaybackStatePersistenceManager(repo, scope)

@Provides
@Singleton
fun provideAutoplayItemProvider(holodexRepository: HolodexRepository): AutoplayItemProvide
    return AutoplayItemProvider(holodexRepository)
}

@Provides
@Singleton
fun provideContinuationManager(
    holodexRepository: HolodexRepository,
    userPreferencesRepository: UserPreferencesRepository,
    autoplayItemProvider: AutoplayItemProvider
): ContinuationManager {
    return ContinuationManager(holodexRepository, userPreferencesRepository, autoplayItemP
}

@Provides
@Singleton
fun providePreloadConfig(): PreloadConfiguration = PreloadConfiguration()

@Provides
@Singleton
```

```
    fun providePreloadStatusController(
        queueManager: PlaybackQueueManager,
        config: PreloadConfiguration
    ): PreloadStatusController {
        return PreloadStatusController(
            getCurrentIndex = { queueManager.playbackQueueFlow.value.currentIndex },
            preloadDurationMs = config.preloadDurationMs
        )
    }
}

// File: java\com\example\holodex\di\Qualifiers.kt
// File: java\com\example\holodex\di\Qualifiers.kt
package com.example.holodex.di

import javax.inject.Qualifier

@Qualifier
@Retention(AnnotationRetention.BINARY)
annotation class ApplicationScope

@Qualifier
@Retention(AnnotationRetention.BINARY)
annotation class PublicClient

@Qualifier
@Retention(AnnotationRetention.BINARY)
annotation class AuthenticatedClient

@Qualifier
@Retention(AnnotationRetention.BINARY)
annotation class DownloadCache

@Qualifier
@Retention(AnnotationRetention.BINARY)
annotation class MediaCache

@Qualifier
@Retention(AnnotationRetention.BINARY)
annotation class UpstreamDataSource

@Qualifier
@Retention(AnnotationRetention.BINARY)
annotation class HolodexHttpClient

@Qualifier
@Retention(AnnotationRetention.BINARY)
annotation class MusicdexHttpClient

@Qualifier
@Retention(AnnotationRetention.BINARY)
annotation class AuthenticatedMusicdexHttpClient

// File: java\com\example\holodex\di\RepositoryModule.kt
// File: java\com\example\holodex\di\RepositoryModule.kt
```

```
package com.example.holodex.di

import android.content.Context
import android.content.SharedPreferences
import androidx.annotation.OptIn
import androidx.media3.common.util.UnstableApi
import androidx.media3.datasource.DataSource
import androidx.media3.datasource.DefaultHttpDataSource
import com.example.holodex.auth.AuthRepository
import com.example.holodex.auth.TokenManager
import com.example.holodex.data.api.AuthenticatedMusicdexApiService
import com.example.holodex.data.api.HolodexApiService
import com.example.holodex.data.api.MusicdexApiService
import com.example.holodex.data.cache.BrowseListCache
import com.example.holodex.data.cache.SearchListCache
import com.example.holodex.data.db.AppDatabase
import com.example.holodex.data.db.DiscoveryDao
import com.example.holodex.data.db.FavoriteChannelDao
import com.example.holodex.data.db.HistoryDao
import com.example.holodex.data.db.LikedItemDao
import com.example.holodex.data.db.LocalDao
import com.example.holodex.data.db.PlaylistDao
import com.example.holodex.data.db.StarredPlaylistDao
import com.example.holodex.data.db.SyncMetadataDao
import com.example.holodex.data.db.VideoDao
import com.example.holodex.data.repository.DownloadRepository
import com.example.holodex.data.repository.DownloadRepositoryImpl
import com.example.holodex.data.repository.HolodexRepository
import com.example.holodex.data.repository.LocalRepository
import com.example.holodex.data.repository.SearchHistoryRepository
import com.example.holodex.data.repository.SharedPreferencesSearchHistoryRepository
import com.example.holodex.data.repository.UserPreferencesRepository
import com.example.holodex.data.repository.YouTubeStreamRepository
import com.example.holodex.data.repository.userPreferencesDataStore
import com.google.gson.Gson
import dagger.Binds
import dagger.Module
import dagger.Provides
import dagger.hilt.InstallIn
import dagger.hilt.android.qualifiers.ApplicationContext
import dagger.hilt.components.SingletonComponent
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers
import net.openid.appauth.AuthorizationService
import javax.inject.Singleton

@Module
@InstallIn(SingletonComponent::class)
abstract class RepositoryModule {

    @Binds
    @Singleton
    abstract fun bindSearchHistoryRepository(
        impl: SharedPreferencesSearchHistoryRepository
```

```
): SearchHistoryRepository

@Binds
@Singleton
@UnstableApi
abstract fun bindDownloadRepository(
    impl: DownloadRepositoryImpl
): DownloadRepository

companion object {

    @Provides
    @Singleton
    fun provideHolodexRepository(
        // --- The existing dependencies are correct ---
        holodexApiService: HolodexApiService,
        musicdexApiService: MusicdexApiService,
        authenticatedMusicdexApiService: AuthenticatedMusicdexApiService,
        discoveryDao: DiscoveryDao,
        browseListCache: BrowseListCache,
        searchListCache: SearchListCache,
        videoDao: VideoDao,
        likedItemDao: LikedItemDao,
        playlistDao: PlaylistDao,
        appDatabase: AppDatabase,
        historyDao: HistoryDao,
        favoriteChannelDao: FavoriteChannelDao,
        syncMetadataDao: SyncMetadataDao,
        starredPlaylistDao: StarredPlaylistDao,
        tokenManager: TokenManager,

        @ApplicationScope applicationScope: CoroutineScope
    ): HolodexRepository {
        return HolodexRepository(
            holodexApiService,
            musicdexApiService,
            authenticatedMusicdexApiService,
            discoveryDao,
            browseListCache,
            searchListCache,
            videoDao,
            likedItemDao,
            playlistDao,
            appDatabase,
            Dispatchers.IO,
            historyDao,
            favoriteChannelDao,
            syncMetadataDao,
            starredPlaylistDao,
            tokenManager,
            applicationScope
        )
    }

    @Provides
```

```
        @Singleton
        fun provideYouTubeStreamRepository(
            sharedPreferences: SharedPreferences,
        ): YouTubeStreamRepository {
            return YouTubeStreamRepository(sharedPreferences)
        }

        @Provides
        @Singleton
        fun provideAuthRepository(
            holodexApiService: HolodexApiService,
            authService: AuthorizationService
        ): AuthRepository {
            return AuthRepository(holodexApiService, authService)
        }

        @Provides
        @Singleton
        fun provideUserPreferencesRepository(@ApplicationContext context: Context): UserPrefer
            return UserPreferencesRepository(context.userPreferencesDataStore)
        }
        @Provides
        @Singleton
        fun provideLocalRepository(localDao: com.example.holodex.data.db.LocalDao): LocalRepos
            return LocalRepository(localDao as LocalDao)
        }
        @Provides
        @Singleton
        fun provideSharedPreferencesSearchHistoryRepository(
            sharedPreferences: SharedPreferences,
            gson: Gson
        ): SharedPreferencesSearchHistoryRepository {
            return SharedPreferencesSearchHistoryRepository(sharedPreferences, gson, Dispatche
        }

        @OptIn(UnstableApi::class)
        @Provides
        @Singleton
        @UpstreamDataSource
        fun provideUpstreamDataSourceFactory(): DataSource.Factory {
            return DefaultHttpDataSource.Factory()
                .setUserAgent("HolodexAppDownloader/1.0")
                .setConnectTimeoutMs(30000)
                .setReadTimeoutMs(30000)
                .setAllowCrossProtocolRedirects(true)
        }
    }
}

// File: java\com\example\holodex\di\SyncModule.kt
// File: java/com/example/holodex/di/SyncModule.kt (MODIFIED)
package com.example.holodex.di

import com.example.holodex.background.FavoriteChannelSynchronizer
import com.example.holodex.background.HistorySynchronizer
```

```
import com.example.holodex.background.ISynchronizer
import com.example.holodex.background.LikesSynchronizer
import com.example.holodex.background.PlaylistSynchronizer
import com.example.holodex.background.StarredPlaylistSynchronizer
import dagger.Binds
import dagger.Module
import dagger.hilt.InstallIn
import dagger.hilt.components.SingletonComponent
import dagger.multibindings.IntoSet

@Module
@InstallIn(SingletonComponent::class)
abstract class SyncModule {

    @Binds
    @IntoSet
    abstract fun bindLikesSynchronizer(impl: LikesSynchronizer): ISynchronizer

    @Binds
    @IntoSet
    abstract fun bindPlaylistSynchronizer(impl: PlaylistSynchronizer): ISynchronizer

    @Binds
    @IntoSet
    abstract fun bindFavoriteChannelSynchronizer(impl: FavoriteChannelSynchronizer): ISynchron

    @Binds
    @IntoSet
    abstract fun bindStarredPlaylistSynchronizer(impl: StarredPlaylistSynchronizer): ISynchron

    @Binds
    @IntoSet
    abstract fun bindHistorySynchronizer(impl: HistorySynchronizer): ISynchronizer
}

// File: java\com\example\holodex\di\UseCaseModule.kt
// File: java/com/example/holodex/di/UseCaseModule.kt
package com.example.holodex.di

import com.example.holodex.data.repository.HolodexRepository
import com.example.holodex.playback.domain.repository.PlaybackRepository
import com.example.holodex.playback.domain.repository.PlaybackStateRepository
import com.example.holodex.playback.domain.repository.StreamResolverRepository
import com.example.holodex.playback.domain.usecase.AddItemToQueueUseCase
import com.example.holodex.playback.domain.usecase.AddItemsToQueueUseCase
import com.example.holodex.playback.domain.usecase.AddOrFetchAndAddUseCase
import com.example.holodex.playback.domain.usecase.ClearQueueUseCase
import com.example.holodex.playback.domain.usecase.GetPlayerSessionIdUseCase
import com.example.holodex.playback.domain.usecase.LoadPlaybackStateUseCase
import com.example.holodex.playback.domain.usecase.ObserveCurrentPlayingItemUseCase
import com.example.holodex.playback.domain.usecase.ObservePlaybackProgressUseCase
import com.example.holodex.playback.domain.usecase.ObservePlaybackQueueUseCase
import com.example.holodex.playback.domain.usecase.ObservePlaybackStateUseCase
import com.example.holodex.playback.domain.usecase.PausePlaybackUseCase
import com.example.holodex.playback.domain.usecase.PlayItemsUseCase
```

```
import com.example.holodex.playback.domain.usecase.ReleasePlaybackResourcesUseCase
import com.example.holodex.playback.domain.usecase.RemoveItemFromQueueUseCase
import com.example.holodex.playback.domain.usecase.ReorderQueueItemUseCase
import com.example.holodex.playback.domain.usecase.ResolveStreamUrlUseCase
import com.example.holodex.playback.domain.usecase.ResumePlaybackUseCase
import com.example.holodex.playback.domain.usecase.SavePlaybackStateUseCase
import com.example.holodex.playback.domain.usecase.SeekPlaybackUseCase
import com.example.holodex.playback.domain.usecase.SetRepeatModeUseCase
import com.example.holodex.playback.domain.usecase.SetScrubbingUseCase
import com.example.holodex.playback.domain.usecase.SetShuffleModeUseCase
import com.example.holodex.playback.domain.usecase.SkipToNextItemUseCase
import com.example.holodex.playback.domain.usecase.SkipToPreviousItemUseCase
import com.example.holodex.playback.domain.usecase.SkipToQueueItemUseCase
import dagger.Module
import dagger.Provides
import dagger.hilt.InstallIn
import dagger.hilt.components.SingletonComponent

@Module
@InstallIn(SingletonComponent::class)
object UseCaseModule {

    @Provides
    fun providePlayItemsUseCase(repo: PlaybackRepository) = PlayItemsUseCase(repo)
    @Provides
    fun providePausePlaybackUseCase(repo: PlaybackRepository) = PausePlaybackUseCase(repo)
    @Provides
    fun provideResumePlaybackUseCase(repo: PlaybackRepository) = ResumePlaybackUseCase(repo)
    @Provides
    fun provideSeekPlaybackUseCase(repo: PlaybackRepository) = SeekPlaybackUseCase(repo)
    @Provides
    fun provideSkipToNextItemUseCase(repo: PlaybackRepository) = SkipToNextItemUseCase(repo)
    @Provides
    fun provideSkipToPreviousItemUseCase(repo: PlaybackRepository) = SkipToPreviousItemUseCase
    @Provides
    fun provideSetRepeatModeUseCase(repo: PlaybackRepository) = SetRepeatModeUseCase(repo)
    @Provides
    fun provideSetShuffleModeUseCase(repo: PlaybackRepository) = SetShuffleModeUseCase(repo)
    @Provides
    fun provideSetScrubbingUseCase(repo: PlaybackRepository) = SetScrubbingUseCase(repo)
    @Provides
    fun provideReleasePlaybackResourcesUseCase(repo: PlaybackRepository) =
        ReleasePlaybackResourcesUseCase(repo)

    @Provides
    fun provideGetPlayerSessionIdUseCase(repo: PlaybackRepository) = GetPlayerSessionIdUseCase

    @Provides
    fun provideObservePlaybackStateUseCase(repo: PlaybackRepository) =
        ObservePlaybackStateUseCase(repo)

    @Provides
    fun provideObservePlaybackProgressUseCase(repo: PlaybackRepository) =
        ObservePlaybackProgressUseCase(repo)
```

```
    @Provides
    fun provideObserveCurrentPlayingItemUseCase(repo: PlaybackRepository) =
        ObserveCurrentPlayingItemUseCase(repo)

    @Provides
    fun provideObservePlaybackQueueUseCase(repo: PlaybackRepository) =
        ObservePlaybackQueueUseCase(repo)

    @Provides
    fun provideAddItemToQueueUseCase(repo: PlaybackRepository) = AddItemToQueueUseCase(repo)
    @Provides
    fun provideAddItemsToQueueUseCase(repo: PlaybackRepository) = AddItemsToQueueUseCase(repo)
    @Provides
    fun provideRemoveItemFromQueueUseCase(repo: PlaybackRepository) =
        RemoveItemFromQueueUseCase(repo)

    @Provides
    fun provideReorderQueueItemUseCase(repo: PlaybackRepository) = ReorderQueueItemUseCase(rep
    @Provides
    fun provideClearQueueUseCase(repo: PlaybackRepository) = ClearQueueUseCase(repo)
    @Provides
    fun provideSkipToQueueItemUseCase(repo: PlaybackRepository) = SkipToQueueItemUseCase(repo)

    @Provides
    fun provideLoadPlaybackStateUseCase(repo: PlaybackStateRepository) =
        LoadPlaybackStateUseCase(repo)

    @Provides
    fun provideSavePlaybackStateUseCase(repo: PlaybackStateRepository) =
        SavePlaybackStateUseCase(repo)

    @Provides
    fun provideResolveStreamUrlUseCase(repo: StreamResolverRepository) =
        ResolveStreamUrlUseCase(repo)

    @Provides
    fun provideAddOrFetchAndAddUseCase(
        repo: HolodexRepository,
        addItemsUseCase: AddItemsToQueueUseCase
    ) = AddOrFetchAndAddUseCase(repo, addItemsUseCase)
}

// File: java\com\example\holodex\extractor\DownloaderImpl.kt
// Location: com.example.holodex.extractor/DownloaderImpl.kt
package com.example.holodex.extractor

import okhttp3.Headers.Companion.toHeaders
import okhttp3.OkHttpClient
import okhttp3.RequestBody.Companion.toRequestBody
import org.schabi.newpipe.extractor.downloader.Downloader
import org.schabi.newpipe.extractor.downloader.Request
import org.schabi.newpipe.extractor.downloader.Response // Ensure this is org.schabi.newpipe.e
import org.schabi.newpipe.extractor.exceptions.ReCaptchaException
import java.io.IOException
```

```
class DownloaderImpl(private val okHttpClient: OkHttpClient) : Downloader() {

    @Throws(IOException::class, ReCaptchaException::class)
    override fun execute(request: Request): Response {
        val okHttpRequestBuilder = okhttp3.Request.Builder()
            .url(request.url())
            .method(request.httpMethod(), request.dataToSend()?.toRequestBody(null))

        for ((headerName, headerValueList) in request.headers()) {
            if (headerValueList.size > 1) {
                headerValueList.forEach { headerValue ->
                    okHttpRequestBuilder.addHeader(headerName, headerValue)
                }
            } else if (headerValueList.size == 1) {
                okHttpRequestBuilder.header(headerName, headerValueList[0])
            }
        }

        val call = okHttpClient.newCall(okHttpRequestBuilder.build())
        val okHttpResponse = call.execute() // This is a synchronous call

        // It's crucial to read the body string only once if you need to inspect it AND pass i
        // If the body is very large, this could be memory inefficient.
        // For ReCaptcha detection, we often only need to check a small part or rely on status
        // However, many ReCaptcha pages are full HTML, so checking content is common.
        val responseBodyString = okHttpResponse.body?.string() // Read body once

        // Simplified ReCaptcha check
        if (okHttpResponse.code == 429 || (responseBodyString != null &&
                    (responseBodyString.contains("consent.youtube.com", ignoreCase = true) ||
                            responseBodyString.contains("??????????", ignoreCase = true) ||
                            responseBodyString.contains("before you continue to youtube", igno
                            responseBodyString.contains("/sorry/index?continue=", ignoreCase =
                            responseBodyString.contains("www.google.com/recaptcha", ignoreCase
                            responseBodyString.contains("??? ??????????, ???????????, ?? ?? ??
                            ))) {
            okHttpResponse.close() // Ensure response is closed if not using its body stream
            throw ReCaptchaException("ReCaptcha Challenge Found", okHttpResponse.request.url.t
        }

        // Pass the already read responseBodyString to the Response constructor
        return Response(
            okHttpResponse.code,
            okHttpResponse.message,
            okHttpResponse.headers.toMultimap(), // Convert OkHttp Headers to Map<String, List
            responseBodyString,                  // Pass the String body directly
            okHttpResponse.request.url.toString()
        )
    }

    // As per your diff, the getCookies/setCookies overrides are removed
    // as they are not part of the current Downloader base class.
    // Cookie management, if needed, should be handled via OkHttp's CookieJar
    // configured on the OkHttpClient instance passed to this DownloaderImpl.
}
```

```
// File: java\com\example\holodex\playback\PlaybackRequestManager.kt
// File: java/com/example/holodex/playback/PlaybackRequestManager.kt
package com.example.holodex.playback

import com.example.holodex.playback.domain.model.PlaybackItem
import kotlinx.coroutines.flow.MutableSharedFlow
import kotlinx.coroutines.flow.SharedFlow
import kotlinx.coroutines.flow.asSharedFlow

data class PlaybackRequestData(
    val items: List<PlaybackItem>,
    val startIndex: Int = 0,
    val startPositionSec: Long = 0L,
    val shouldShuffle: Boolean = false
)

/**
 * A singleton class to centralize playback requests from any part of the app.
 */
class PlaybackRequestManager {
    private val _playbackRequest = MutableSharedFlow<PlaybackRequestData>()
    val playbackRequest: SharedFlow<PlaybackRequestData> = _playbackRequest.asSharedFlow()

    suspend fun submitPlaybackRequest(
        items: List<PlaybackItem>,
        startIndex: Int = 0,
        startPositionSec: Long = 0L,
        shouldShuffle: Boolean = false
    ) {
        if (items.isEmpty()) return
        _playbackRequest.emit(
            PlaybackRequestData(items, startIndex, startPositionSec, shouldShuffle)
        )
    }
}

// File: java\com\example\holodex\playback\data\mapper\MediaItemMapper.kt
// File: java/com/example/holodex/playback/data/mapper/MediaItemMapper.kt
package com.example.holodex.playback.data.mapper

import android.content.Context
import android.content.SharedPreferences
import android.os.Bundle
import androidx.core.net.toUri
import androidx.media3.common.C
import androidx.media3.common.MediaItem
import androidx.media3.common.MediaMetadata
import com.example.holodex.R
import com.example.holodex.playback.domain.model.PersistedPlaybackItem
import com.example.holodex.playback.domain.model.PlaybackItem
import com.example.holodex.util.getHighResArtworkUrl
import com.example.holodex.viewmodel.AppPreferenceConstants
import timber.log.Timber
import java.util.concurrent.TimeUnit
```

```
internal const val EXTRA_KEY_HOLODEX_VIDEO_ID = "com.example.holodex.EXTRA_VIDEO_ID"
internal const val EXTRA_KEY_HOLODEX_SONG_ID = "com.example.holodex.EXTRA_SONG_ID"
internal const val EXTRA_KEY_HOLODEX_SERVER_UUID = "com.example.holodex.EXTRA_SERVER_UUID"
internal const val EXTRA_KEY_ORIGINAL_DURATION_SEC = "com.example.holodex.EXTRA_DURATION_SEC"
internal const val EXTRA_KEY_ARTIST_TEXT = "com.example.holodex.EXTRA_ARTIST_TEXT"
internal const val EXTRA_KEY_ALBUM_TEXT = "com.example.holodex.EXTRA_ALBUM_TEXT"
internal const val EXTRA_KEY_DESCRIPTION_TEXT = "com.example.holodex.EXTRA_DESCRIPTION_TEXT"
internal const val EXTRA_KEY_HOLODEX_CHANNEL_ID = "com.example.holodex.EXTRA_CHANNEL_ID"
private const val MAPPER_TAG = "MediaItemMapper"
private val PLACEHOLDER_URI = "placeholder://unresolved".toUri()


class MediaItemMapper(
    private val context: Context,
    private val sharedPreferences: SharedPreferences
) {

    fun toMedia3MediaItem(playbackItem: PlaybackItem): MediaItem? {
        Timber.tag(MAPPER_TAG).d("toMedia3MediaItem: Mapping PlaybackItem ID '${playbackItem.i

        if (playbackItem.streamUri.isNullOrBlank()) {
            Timber.e("CRITICAL: PlaybackItem ${playbackItem.id} has no streamUri. Cannot creat
            return null
        }

        val extras = Bundle().apply {
            putString(EXTRA_KEY_HOLODEX_VIDEO_ID, playbackItem.videoId)
            playbackItem.songId?.let { putString(EXTRA_KEY_HOLODEX_SONG_ID, it) }
            playbackItem.serverUuid?.let { putString(EXTRA_KEY_HOLODEX_SERVER_UUID, it) }
            putLong(EXTRA_KEY_ORIGINAL_DURATION_SEC, playbackItem.durationSec)
            putString(EXTRA_KEY_ARTIST_TEXT, playbackItem.artistText)
            playbackItem.albumText?.let { putString(EXTRA_KEY_ALBUM_TEXT, it) }
            playbackItem.description?.let { putString(EXTRA_KEY_DESCRIPTION_TEXT, it) }
            putString(EXTRA_KEY_HOLODEX_CHANNEL_ID, playbackItem.channelId)
        }

        val imageQualityPref = sharedPreferences.getString(AppPreferenceConstants.PREF_IMAGE_Q
            ?: AppPreferenceConstants.IMAGE_QUALITY_AUTO
        val highResArtworkUriString = getHighResArtworkUrl(playbackItem.artworkUri, imageQuali

        val mediaMetadata = MediaMetadata.Builder()
            .setTitle(playbackItem.title.ifBlank { context.getString(R.string.unknown_title) }
            .setArtist(playbackItem.artistText.ifBlank { context.getString(R.string.unknown_ar
            .setAlbumTitle(playbackItem.albumText ?: playbackItem.title)
            .setArtworkUri(highResArtworkUriString?.toUri())
            .setExtras(extras)
            .build()

        val mediaItemBuilder = MediaItem.Builder()
            .setMediaId(playbackItem.id)
            .setMediaMetadata(mediaMetadata)

        if (!playbackItem.streamUri.isNullOrBlank()) {
            mediaItemBuilder.setUri(playbackItem.streamUri)
```

```
    } else {
        Timber.e("CRITICAL: PlaybackItem ${playbackItem.id} converted to MediaItem with no
    }

    val isLocalFile = playbackItem.streamUri?.startsWith("content://") == true

    if (!isLocalFile && playbackItem.clipStartSec != null && playbackItem.clipEndSec != nu
        // This is a stream, apply clipping as before.
        mediaItemBuilder.setClippingConfiguration(
            MediaItem.ClippingConfiguration.Builder()
                .setStartPositionMs(playbackItem.clipStartSec * 1000L)
                .setEndPositionMs(playbackItem.clipEndSec * 1000L)
                .setRelativeToDefaultPosition(false)
                .build()
        )
        Timber.d("Applied STREAM clipping to MediaItem ${playbackItem.id}: Start: ${playba
    } else {
        // This is either a local file or a full video stream, no clipping needed.
        Timber.d("Skipping clipping for MediaItem ${playbackItem.id}. Is local file: $isLo
    }

    return mediaItemBuilder.build()
}

fun toPlaceholderMediaItem(playbackItem: PlaybackItem): MediaItem {
    Timber.tag(MAPPER_TAG).d("toPlaceholderMediaItem: Creating placeholder for '${playback

    val extras = Bundle().apply {
        putString(EXTRA_KEY_HOLODEX_VIDEO_ID, playbackItem.videoId)
        playbackItem.songId?.let { putString(EXTRA_KEY_HOLODEX_SONG_ID, it) }
        playbackItem.serverUuid?.let { putString(EXTRA_KEY_HOLODEX_SERVER_UUID, it) }
        putLong(EXTRA_KEY_ORIGINAL_DURATION_SEC, playbackItem.durationSec)
        putString(EXTRA_KEY_ARTIST_TEXT, playbackItem.artistText)
        playbackItem.albumText?.let { putString(EXTRA_KEY_ALBUM_TEXT, it) }
        playbackItem.description?.let { putString(EXTRA_KEY_DESCRIPTION_TEXT, it) }
        putString(EXTRA_KEY_HOLODEX_CHANNEL_ID, playbackItem.channelId)
    }

    val imageQualityPref = sharedPreferences.getString(AppPreferenceConstants.PREF_IMAGE_Q
        ?: AppPreferenceConstants.IMAGE_QUALITY_AUTO
    val highResArtworkUriString = getHighResArtworkUrl(playbackItem.artworkUri, imageQuali

    val mediaMetadata = MediaMetadata.Builder()
        .setTitle(playbackItem.title.ifBlank { context.getString(R.string.unknown_title) }
        .setArtist(playbackItem.artistText.ifBlank { context.getString(R.string.unknown_ar
        .setAlbumTitle(playbackItem.albumText ?: playbackItem.title)
        .setArtworkUri(highResArtworkUriString?.toUri())
        .setExtras(extras)
        .build()

    return MediaItem.Builder()
        .setMediaId(playbackItem.id)
        .setMediaMetadata(mediaMetadata)
        .setUri(PLACEHOLDER_URI)
        .build()
```

```
}

fun toPlaybackItem(mediaItem: MediaItem): PlaybackItem? {
    val mediaId = mediaItem.mediaId
    if (mediaId.isBlank()) {
        Timber.w("Cannot convert MediaItem to PlaybackItem: mediaId is null or blank.")
        return null
    }

    val metadata = mediaItem.mediaMetadata
    val extras = metadata.extras ?: Bundle.EMPTY

    val durationSecFromExtras = extras.getLong(EXTRA_KEY_ORIGINAL_DURATION_SEC, -1L)

    val clipStartSecFromMediaItem =
        if (mediaItem.clippingConfiguration.startPositionMs != C.TIME_UNSET) {
            TimeUnit.MILLISECONDS.toSeconds(mediaItem.clippingConfiguration.startPositionM
        } else null
    val clipEndSecFromMediaItem =
        if (mediaItem.clippingConfiguration.endPositionMs != C.TIME_UNSET) {
            TimeUnit.MILLISECONDS.toSeconds(mediaItem.clippingConfiguration.endPositionMs)
        } else null

    val finalDurationSec = if (durationSecFromExtras >= 0) {
        durationSecFromExtras
    } else {
        Timber.w("MediaItem $mediaId: Could not determine duration from extras. Defaulting
        0L
    }

    return PlaybackItem(
        id = mediaId,
        videoId = extras.getString(EXTRA_KEY_HOLODEX_VIDEO_ID) ?: "unknown_video_id",
        serverUuid = extras.getString(EXTRA_KEY_HOLODEX_SERVER_UUID),
        songId = extras.getString(EXTRA_KEY_HOLODEX_SONG_ID),
        title = metadata.title?.toString() ?: context.getString(R.string.unknown_title),
        artistText = extras.getString(EXTRA_KEY_ARTIST_TEXT) ?: metadata.artist?.toString(
        ?: context.getString(R.string.unknown_artist),
        albumText = extras.getString(EXTRA_KEY_ALBUM_TEXT) ?: metadata.albumTitle?.toStrin
        artworkUri = metadata.artworkUri?.toString(),
        durationSec = finalDurationSec,
        streamUri = mediaItem.localConfiguration?.uri?.toString(),
        clipStartSec = clipStartSecFromMediaItem,
        clipEndSec = clipEndSecFromMediaItem,
        description = extras.getString(EXTRA_KEY_DESCRIPTION_TEXT),
        channelId = extras.getString(EXTRA_KEY_HOLODEX_CHANNEL_ID) ?: "unknown_channel_id"
        originalArtist = null // This info is not typically passed through MediaItem
    )
}

fun toPersistedPlaybackItem(playbackItem: PlaybackItem): PersistedPlaybackItem {
    return PersistedPlaybackItem(
        id = playbackItem.id,
        videoId = playbackItem.videoId,
        songId = playbackItem.serverUuid, // Persist the serverUuid in the songId field
```

```
            title = playbackItem.title,
            artistText = playbackItem.artistText,
            albumText = playbackItem.albumText,
            artworkUri = playbackItem.artworkUri,
            durationSec = playbackItem.durationSec,
            description = playbackItem.description,
            channelId = playbackItem.channelId,
            clipStartSec = playbackItem.clipStartSec,
            clipEndSec = playbackItem.clipEndSec
        )
    }

    fun toPlaybackItem(persistedPlaybackItem: PersistedPlaybackItem): PlaybackItem {
        return PlaybackItem(
            id = persistedPlaybackItem.id,
            videoId = persistedPlaybackItem.videoId,
            serverUuid = persistedPlaybackItem.songId,
            songId = persistedPlaybackItem.songId,
            title = persistedPlaybackItem.title,
            artistText = persistedPlaybackItem.artistText,
            albumText = persistedPlaybackItem.albumText,
            artworkUri = persistedPlaybackItem.artworkUri,
            durationSec = persistedPlaybackItem.durationSec,
            streamUri = null,
            description = persistedPlaybackItem.description,
            channelId = persistedPlaybackItem.channelId,
            clipStartSec = persistedPlaybackItem.clipStartSec,
            clipEndSec = persistedPlaybackItem.clipEndSec,
            originalArtist = null
        )
    }
}


// File: java\com\example\holodex\playback\data\mapper\PersistedPlaybackStateMapper.kt
// File: java/com/example/holodex/playback/data/mapper/PersistedPlaybackStateMapper.kt
package com.example.holodex.playback.data.mapper

import com.example.holodex.playback.data.model.PersistedPlaybackItemEntity
import com.example.holodex.playback.data.model.PersistedPlaybackStateEntity
import com.example.holodex.playback.domain.model.PersistedPlaybackData
import com.example.holodex.playback.domain.model.PersistedPlaybackItem

fun PersistedPlaybackData.toEntities(): Pair<PersistedPlaybackStateEntity, List<PersistedPlayb
    val stateEntity = PersistedPlaybackStateEntity(
        queueIdentifier = this.queueId,
        currentIndex = this.currentIndex,
        currentPositionSec = this.currentPositionSec * 1000L,
        currentItemId = this.currentItemId,
        repeatMode = this.repeatMode,
        shuffleMode = this.shuffleMode,
        shuffledItemIds = this.shuffledQueueItemIds,
        shuffleOrderVersion = 1
    )
```

```
    val itemEntities = this.queueItems.mapIndexed { order, domainItem ->
        PersistedPlaybackItemEntity(
            ownerQueueIdentifier = this.queueId,
            itemPlaybackId = domainItem.id,
            videoId = domainItem.videoId,
            songId = domainItem.songId,
            title = domainItem.title,
            artistText = domainItem.artistText,
            albumText = domainItem.albumText,
            artworkUri = domainItem.artworkUri,
            durationMs = domainItem.durationSec * 1000L,
            itemOrder = order,
            description = domainItem.description,
            channelId = domainItem.channelId,
            clipStartMs = domainItem.clipStartSec?.let { it * 1000L },
            clipEndMs = domainItem.clipEndSec?.let { it * 1000L }
        )
    }
    return Pair(stateEntity, itemEntities)
}

fun PersistedPlaybackStateEntity.toDomainModel(itemEntities: List<PersistedPlaybackItemEntity>
    val domainItems = itemEntities
        .sortedBy { it.itemOrder }
        .map { entity ->
            PersistedPlaybackItem(
                id = entity.itemPlaybackId,
                videoId = entity.videoId,
                songId = entity.songId,
                title = entity.title,
                artistText = entity.artistText,
                albumText = entity.albumText,
                artworkUri = entity.artworkUri,
                durationSec = entity.durationMs / 1000L,
                description = entity.description,
                channelId = entity.channelId,
                clipStartSec = entity.clipStartMs?.let { it / 1000L },
                clipEndSec = entity.clipEndMs?.let { it / 1000L }
            )
        }

    return PersistedPlaybackData(
        queueId = this.queueIdentifier,
        queueItems = domainItems,
        currentIndex = this.currentIndex,
        currentPositionSec = this.currentPositionSec / 1000L,
        currentItemId = this.currentItemId,
        repeatMode = this.repeatMode,
        shuffleMode = this.shuffleMode,
        shuffledQueueItemIds = this.shuffledItemIds
    )
}

// File: java\com\example\holodex\playback\data\model\PersistedPlaybackItemEntity.kt
// File: java/com/example/holodex/playback/data/model/PersistedPlaybackItemEntity.kt
```

```
package com.example.holodex.playback.data.model

import androidx.room.ColumnInfo
import androidx.room.Entity
import androidx.room.ForeignKey
import androidx.room.Index
import androidx.room.PrimaryKey

@Entity(
    tableName = "persisted_playback_items_table",
    foreignKeys = [ForeignKey(
        entity = PersistedPlaybackStateEntity::class,
        parentColumns = ["queueIdentifier"],
        childColumns = ["ownerQueueIdentifier"],
        onDelete = ForeignKey.CASCADE
    )],
    indices = [Index(value = ["ownerQueueIdentifier"])]
)
data class PersistedPlaybackItemEntity(
    @PrimaryKey(autoGenerate = true) val dbId: Long = 0,
    val ownerQueueIdentifier: String,
    val itemPlaybackId: String,
    val videoId: String,
    val songId: String?,
    val title: String,
    val artistText: String,
    val albumText: String?,
    val artworkUri: String?,
    val durationMs: Long,
    val itemOrder: Int,
    val description: String? = null,
    @ColumnInfo(name = "channel_id") val channelId: String,
    @ColumnInfo(name = "clip_start_ms") val clipStartMs: Long? = null,
    @ColumnInfo(name = "clip_end_ms") val clipEndMs: Long? = null
)

// File: java\com\example\holodex\playback\data\model\PersistedPlaybackStateDao.kt
// File: java/com/example/holodex/playback/data/model/PersistedPlaybackStateDao.kt
package com.example.holodex.playback.data.model

import androidx.room.Dao
import androidx.room.Embedded
import androidx.room.Insert
import androidx.room.OnConflictStrategy
import androidx.room.Query
import androidx.room.Relation
import androidx.room.Transaction

data class PlaybackStateWithItemsTuple(
    @Embedded val state: PersistedPlaybackStateEntity,
    @Relation(
        parentColumn = "queueIdentifier",
        entityColumn = "ownerQueueIdentifier"
    )
    val items: List<PersistedPlaybackItemEntity>
```

```
)

@Dao
interface PersistedPlaybackStateDao {

    @Transaction
    @Query("SELECT * FROM persisted_playback_state_table WHERE queueIdentifier = :targetQueueI
    suspend fun getPlaybackStateWithItems(targetQueueIdentifier: String = "default_queue"): Pl

    @Transaction
    suspend fun savePlaybackStateWithItems(stateEntity: PersistedPlaybackStateEntity, itemEnti
        clearPlaybackStateByQueueId(stateEntity.queueIdentifier)
        clearPlaybackItemsByQueueId(stateEntity.queueIdentifier)
        insertPlaybackState(stateEntity)
        insertPlaybackItems(itemEntities)
    }

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertPlaybackState(state: PersistedPlaybackStateEntity)

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertPlaybackItems(items: List<PersistedPlaybackItemEntity>)

    @Query("DELETE FROM persisted_playback_state_table WHERE queueIdentifier = :targetQueueIde
    suspend fun clearPlaybackStateByQueueId(targetQueueIdentifier: String)

    @Query("DELETE FROM persisted_playback_items_table WHERE ownerQueueIdentifier = :targetQue
    suspend fun clearPlaybackItemsByQueueId(targetQueueIdentifier: String)
}

// File: java\com\example\holodex\playback\data\model\PersistedPlaybackStateEntity.kt
// File: java/com/example/holodex/playback/data/model/PersistedPlaybackStateEntity.kt
package com.example.holodex.playback.data.model

import androidx.room.ColumnInfo
import androidx.room.Entity
import androidx.room.PrimaryKey
import com.example.holodex.playback.domain.model.DomainRepeatMode
import com.example.holodex.playback.domain.model.DomainShuffleMode

@Entity(tableName = "persisted_playback_state_table")
data class PersistedPlaybackStateEntity(
    @PrimaryKey
    val queueIdentifier: String,
    val currentIndex: Int,
    val currentPositionSec: Long,
    @ColumnInfo(name = "current_item_id")
    val currentItemId: String?,
    val repeatMode: DomainRepeatMode,
    val shuffleMode: DomainShuffleMode,
    @ColumnInfo(name = "shuffled_item_ids")
    val shuffledItemIds: List<String>? = null,
    @ColumnInfo(name = "shuffle_order_version", defaultValue = "1")
    val shuffleOrderVersion: Int = 1
)
```

```
// File: java\com\example\holodex\playback\data\persistence\PlaybackStatePersistenceManager.kt
// File: java/com/example/holodex/playback/data/persistence/PlaybackStatePersistenceManager.kt
package com.example.holodex.playback.data.persistence

import com.example.holodex.playback.domain.model.PersistedPlaybackData
import com.example.holodex.playback.domain.repository.PlaybackStateRepository
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Job
import kotlinx.coroutines.delay
import kotlinx.coroutines.isActive
import kotlinx.coroutines.launch
import timber.log.Timber

class PlaybackStatePersistenceManager(
    private val playbackStateRepository: PlaybackStateRepository,
    private val externalScope: CoroutineScope
) {
    companion object {
        private const val TAG = "PlaybackStatePersistMgr"
        private const val DEFAULT_SAVE_DELAY_MS = 750L
    }

    private var saveStateJob: Job? = null

    suspend fun saveState(data: PersistedPlaybackData) {
        Timber.d("$TAG: Saving state directly. Queue ID: ${data.queueId}, Items: ${data.queueI
        try {
            playbackStateRepository.saveState(data)
        } catch (e: Exception) {
            Timber.e(e, "$TAG: Error during direct state save.")
        }
    }

    suspend fun loadState(): PersistedPlaybackData? {
        Timber.d("$TAG: Loading state.")
        return try {
            playbackStateRepository.loadState()
        } catch (e: Exception) {
            Timber.e(e, "$TAG: Error loading state.")
            null
        }
    }

    suspend fun clearState() {
        Timber.d("$TAG: Clearing state.")
        try {
            saveStateJob?.cancel()
            playbackStateRepository.clearState()
        } catch (e: Exception) {
            Timber.e(e, "$TAG: Error clearing state.")
        }
    }

    fun scheduleSave(data: PersistedPlaybackData, delayMs: Long = DEFAULT_SAVE_DELAY_MS) {
```

```
        saveStateJob?.cancel()
        saveStateJob = externalScope.launch {
            delay(delayMs)
            if (isActive) {
                Timber.d("$TAG: Scheduled save executing after delay. Queue ID: ${data.queueId
                saveState(data)
            } else {
                Timber.d("$TAG: Scheduled save cancelled before execution.")
            }
        }
        Timber.v("$TAG: State save scheduled with delay: $delayMs ms.")
    }
}

// File: java\com\example\holodex\playback\data\preload\PreloadConfiguration.kt
// File: java/com/example/holodex/playback/data/preload/PreloadConfiguration.kt
package com.example.holodex.playback.data.preload

data class PreloadConfiguration(
    val preloadDurationMs: Long = 10_000L,
    val maxConcurrentPreloads: Int = 2,
    val isEnabled: Boolean = true
)

// File: java\com\example\holodex\playback\data\preload\PreloadStatusController.kt
// File: java/com/example/holodex/playback/data/preload/PreloadStatusController.kt
package com.example.holodex.playback.data.preload

import androidx.media3.common.util.UnstableApi
import androidx.media3.exoplayer.source.preload.DefaultPreloadManager
import androidx.media3.exoplayer.source.preload.TargetPreloadStatusControl
import timber.log.Timber

@UnstableApi
class PreloadStatusController(
    private val getCurrentIndex: () -> Int,
    private val preloadDurationMs: Long = 10_000L
) : TargetPreloadStatusControl<Int, DefaultPreloadManager.PreloadStatus> {

    companion object {
        private const val TAG = "PreloadStatusController"
    }

    override fun getTargetPreloadStatus(rankingData: Int): DefaultPreloadManager.PreloadStatus
        val currentIndex = getCurrentIndex()
        val ranking = rankingData - currentIndex

        return when (ranking) {
            1 -> {
                Timber.i("$TAG: Preloading next item (index $rankingData) for ${preloadDuratio
                DefaultPreloadManager.PreloadStatus.specifiedRangeLoaded(preloadDurationMs)
            }
            2 -> {
                Timber.i("$TAG: Preloading second item (index $rankingData) for ${preloadDurat
                DefaultPreloadManager.PreloadStatus.specifiedRangeLoaded(preloadDurationMs / 2
```

```
            }
            else -> {
                null
            }
        }
    }
}

// File: java\com\example\holodex\playback\data\queue\PlaybackQueueManager.kt
// File: java/com/example/holodex/playback/data/queue/PlaybackQueueManager.kt
package com.example.holodex.playback.data.queue

import com.example.holodex.playback.domain.model.DomainRepeatMode
import com.example.holodex.playback.domain.model.DomainShuffleMode
import com.example.holodex.playback.domain.model.PlaybackItem
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asStateFlow
import timber.log.Timber

class PlaybackQueueManager(
    private val shuffleOrderProvider: ShuffleOrderProvider
) {
    private val _playbackQueueFlow = MutableStateFlow(PlaybackQueueState())
    val playbackQueueFlow: StateFlow<PlaybackQueueState> = _playbackQueueFlow.asStateFlow()

    companion object {
        private const val TAG = "QUEUE_SHUFFLE_DEBUG"
    }

    fun dispatch(action: QueueAction) {
        val currentState = _playbackQueueFlow.value
        Timber.tag(TAG).i("[QueueManager] dispatch() received Action: ${action::class.simpleNa
        val newState = reduce(currentState, action)
        _playbackQueueFlow.value = newState
    }

    private fun calculateNextIndex(currentState: PlaybackQueueState): Int {
        if (currentState.activeList.isEmpty()) return -1
        return when (currentState.repeatMode) {
            DomainRepeatMode.ONE -> currentState.currentIndex // Repeat one just stays on the
            DomainRepeatMode.ALL -> if (currentState.currentIndex >= currentState.activeList.s
            DomainRepeatMode.NONE -> currentState.currentIndex + 1 // This will go out of boun
        }
    }

    private fun calculatePreviousIndex(currentState: PlaybackQueueState): Int {
        if (currentState.activeList.isEmpty()) return -1
        // In all modes, going previous from the start either goes to the end (if repeating) o
        return if (currentState.currentIndex <= 0) {
            if (currentState.repeatMode == DomainRepeatMode.ALL) currentState.activeList.size
        } else {
            currentState.currentIndex - 1
        }
    }
```

```
fun calculateInsertionIndex(item: PlaybackItem, requestedIndex: Int?): Int {
    val currentState = _playbackQueueFlow.value
    val insertAt = requestedIndex ?: currentState.originalList.size

    return if (currentState.shuffleMode == DomainShuffleMode.ON) {
        // In shuffle mode, new items are always added to the end of the active (shuffled)
        currentState.activeList.size
    } else {
        // In normal mode, add it where requested in the active (original) list.
        insertAt
    }
}
private fun reduce(currentState: PlaybackQueueState, action: QueueAction): PlaybackQueueSt
    return when (action) {
        is QueueAction.SetQueue -> {
            val originalList = action.items
            var activeList = originalList
            var currentIndex = action.startIndex
            var shuffleMode = action.restoredShuffleMode ?: DomainShuffleMode.OFF
            val repeatMode = action.restoredRepeatMode ?: DomainRepeatMode.NONE

            if (action.shouldShuffle) {
                val itemToStartWith = originalList.getOrNull(action.startIndex)
                if (itemToStartWith != null) {
                    activeList =
                        shuffleOrderProvider.createShuffledList(originalList, itemToStartW
                    currentIndex = 0
                    shuffleMode = DomainShuffleMode.ON
                }
            } else if (shuffleMode == DomainShuffleMode.ON && !action.restoredShuffledList
                activeList = action.restoredShuffledList
            }

            return currentState.copy(
                originalList = originalList,
                activeList = activeList,
                currentIndex = currentIndex,
                shuffleMode = shuffleMode,
                repeatMode = repeatMode,
                transientStartPositionMs = action.startPositionMs
            )
        }

        is QueueAction.ToggleShuffle -> {
            val newShuffleMode = if (currentState.shuffleMode == DomainShuffleMode.ON) Dom
            val currentItem = currentState.currentItem ?: return currentState.copy(shuffle

            if (newShuffleMode == DomainShuffleMode.ON) {
                val newActiveList = shuffleOrderProvider.createShuffledList(
                    currentState.originalList,
                    currentItem
                )
                return currentState.copy(
                    activeList = newActiveList,
                    currentIndex = 0,
```

```
                shuffleMode = newShuffleMode
            )
        } else {
            val newActiveList = currentState.originalList
            val newIndex = newActiveList.indexOf(currentItem).coerceAtLeast(0)
            return currentState.copy(
                activeList = newActiveList,
                currentIndex = newIndex,
                shuffleMode = newShuffleMode
            )
        }
    }

    is QueueAction.SetRepeatMode -> currentState.copy(repeatMode = action.repeatMode)
    is QueueAction.AddItem -> {
        val insertAt = action.index ?: currentState.originalList.size
        val newOriginal = currentState.originalList.toMutableList().apply { add(insert
        // If shuffling, add to a random position in the active list (but not at the c
        val newActive = if (currentState.shuffleMode == DomainShuffleMode.ON) {
            currentState.activeList.toMutableList().apply {
                // Add somewhere after the current item
                val randomPosition = if (size > currentState.currentIndex + 1) {
                    (currentState.currentIndex + 1 until size).random()
                } else {
                    size
                }
                add(randomPosition, action.item)
            }
        } else {
            currentState.activeList.toMutableList().apply { add(insertAt, action.item)
        }
        currentState.copy(originalList = newOriginal, activeList = newActive)
    }

    // --- START OF FIX: Add the case for AddItems ---
    is QueueAction.AddItems -> {
        if (action.items.isEmpty()) return currentState

        val insertAt = action.index ?: currentState.originalList.size
        val newOriginal = currentState.originalList.toMutableList().apply { addAll(ins

        val newActive = if (currentState.shuffleMode == DomainShuffleMode.ON) {
            // When adding multiple items to a shuffled queue, just append the new ite
            // (shuffled among themselves) to the end of the active queue.
            currentState.activeList + action.items.shuffled()
        } else {
            currentState.activeList.toMutableList().apply { addAll(insertAt, action.it
        }
        currentState.copy(originalList = newOriginal, activeList = newActive)
    }

    is QueueAction.RemoveItem -> {
        if (action.index !in currentState.activeList.indices) return currentState
        val itemToRemove = currentState.activeList[action.index]
        val newActive = currentState.activeList.toMutableList().apply { removeAt(actio
```

```
            val newOriginal = currentState.originalList.toMutableList().apply { remove(ite
            val newCurrentIndex = when {
                action.index < currentState.currentIndex -> currentState.currentIndex - 1
                action.index == currentState.currentIndex -> if (currentState.currentIndex
                else -> currentState.currentIndex
            }
            currentState.copy(
                originalList = newOriginal,
                activeList = newActive,
                currentIndex = newCurrentIndex
            )
        }

        is QueueAction.SetCurrentIndex -> {
            if (action.newIndex != currentState.currentIndex) {
                currentState.copy(currentIndex = action.newIndex)
            } else {
                currentState
            }
        }

        is QueueAction.ClearQueue -> PlaybackQueueState()

        is QueueAction.UpdateItemInQueue -> {
            return currentState.copy(
                originalList = currentState.originalList.map { if (it.id == action.updated
                activeList = currentState.activeList.map { if (it.id == action.updatedItem
            )
        }
        is QueueAction.ReorderItem -> {
            if (action.fromIndex !in currentState.activeList.indices || action.toIndex !in
                return currentState // Invalid indices, do nothing
            }
            val reorderedList = currentState.activeList.toMutableList().apply {
                add(action.toIndex, removeAt(action.fromIndex))
            }
            // Adjust the current playing index if it was affected by the move
            val newCurrentIndex = when {
                currentState.currentIndex == action.fromIndex -> action.toIndex
                action.fromIndex < currentState.currentIndex && action.toIndex >= currentS
                action.fromIndex > currentState.currentIndex && action.toIndex <= currentS
                else -> currentState.currentIndex
            }
            return currentState.copy(activeList = reorderedList, currentIndex = newCurrent
        }
        is QueueAction.SkipToNext -> {
            val nextIndex = calculateNextIndex(currentState)
            // We only update the index if it's within the bounds of the list.
            // If it goes out of bounds, the repository will see this and know playback ha
            if (nextIndex in currentState.activeList.indices) {
                currentState.copy(currentIndex = nextIndex)
            } else {
                // Let the state reflect that we've gone past the end
                currentState.copy(currentIndex = nextIndex)
            }
```

```
            }
            is QueueAction.SkipToPrevious -> {
                val previousIndex = calculatePreviousIndex(currentState)
                currentState.copy(currentIndex = previousIndex)
            }
            else -> currentState // Placeholder for other actions
        }
    }
}

// File: java\com\example\holodex\playback\data\queue\PlaybackQueueState.kt
// File: java/com/example/holodex/playback/data/queue/PlaybackQueueState.kt
package com.example.holodex.playback.data.queue

import com.example.holodex.playback.domain.model.DomainRepeatMode
import com.example.holodex.playback.domain.model.DomainShuffleMode
import com.example.holodex.playback.domain.model.PlaybackItem

data class PlaybackQueueState(
    val originalList: List<PlaybackItem> = emptyList(),
    val activeList: List<PlaybackItem> = emptyList(),
    val currentIndex: Int = -1,
    val shuffleMode: DomainShuffleMode = DomainShuffleMode.OFF,
    val repeatMode: DomainRepeatMode = DomainRepeatMode.NONE,
    val transientStartPositionMs: Long = 0L
) {
    val currentItem: PlaybackItem?
        get() = activeList.getOrNull(currentIndex)
}

// File: java\com\example\holodex\playback\data\queue\QueueAction.kt
// File: java/com/example/holodex/playback/data/queue/QueueAction.kt
package com.example.holodex.playback.data.queue

import com.example.holodex.playback.domain.model.DomainRepeatMode
import com.example.holodex.playback.domain.model.DomainShuffleMode
import com.example.holodex.playback.domain.model.PlaybackItem

sealed class QueueAction {
    data class SetQueue(
        val items: List<PlaybackItem>,
        val startIndex: Int,
        val startPositionMs: Long = 0L,
        val shouldShuffle: Boolean = false,
        val restoredShuffleMode: DomainShuffleMode? = null,
        val restoredRepeatMode: DomainRepeatMode? = null,
        val restoredShuffledList: List<PlaybackItem>? = null
    ) : QueueAction()

    object ToggleShuffle : QueueAction()
    data class SetRepeatMode(val repeatMode: DomainRepeatMode) : QueueAction()
    data class ReorderItem(val fromIndex: Int, val toIndex: Int) : QueueAction()
    data class AddItem(val item: PlaybackItem, val index: Int?) : QueueAction()
    data class AddItems(val items: List<PlaybackItem>, val index: Int?) : QueueAction()
    data class RemoveItem(val index: Int) : QueueAction()
```

```
    data class SetCurrentIndex(val newIndex: Int) : QueueAction()
    object ClearQueue : QueueAction()
    data class UpdateItemInQueue(val updatedItem: PlaybackItem) : QueueAction()
    object SkipToNext : QueueAction()
    object SkipToPrevious : QueueAction()
}

// File: java\com\example\holodex\playback\data\queue\ShuffleOrderProvider.kt
// File: java/com/example/holodex/playback/data/queue/ShuffleOrderProvider.kt
package com.example.holodex.playback.data.queue

import com.example.holodex.playback.domain.model.PlaybackItem

class ShuffleOrderProvider {
    fun createShuffledList(
        originalItems: List<PlaybackItem>,
        currentItem: PlaybackItem?
    ): List<PlaybackItem> {
        if (originalItems.isEmpty()) return emptyList()

        val mutableList = originalItems.toMutableList()
        val currentIndex = currentItem?.let { originalItems.indexOf(it) } ?: -1

        if (currentIndex >= 0) {
            val current = mutableList.removeAt(currentIndex)
            mutableList.shuffle()
            return listOf(current) + mutableList
        } else {
            mutableList.shuffle()
            return mutableList
        }
    }
}

// File: java\com\example\holodex\playback\data\repository\HolodexStreamResolverRepositoryImpl
// File: java/com/example/holodex/playback/data/repository/HolodexStreamResolverRepositoryImpl
package com.example.holodex.playback.data.repository

import com.example.holodex.data.model.AudioStreamDetails
import com.example.holodex.data.repository.YouTubeStreamRepository
import com.example.holodex.playback.domain.model.StreamDetails
import com.example.holodex.playback.domain.repository.StreamResolverRepository
import javax.inject.Inject
import javax.inject.Singleton

@Singleton
class HolodexStreamResolverRepositoryImpl @Inject constructor(
    private val youtubeStreamRepository: YouTubeStreamRepository
) : StreamResolverRepository {

    override suspend fun resolveStreamUrl(videoId: String): Result<StreamDetails> {
        return try {
            val youtubeResult: Result<AudioStreamDetails> = youtubeStreamRepository.getAudioSt

            youtubeResult.map { oldDetails ->
```

```
                val streamUrl = oldDetails.streamUrl ?: throw IllegalStateException("Stream UR
                StreamDetails(
                    url = streamUrl,
                    format = oldDetails.format,
                    quality = oldDetails.quality
                )
            }
        } catch (e: Exception) {
            Result.failure(e)
        }
    }
}

// File: java\com\example\holodex\playback\data\repository\Media3PlaybackRepositoryImpl.kt
// File: java/com/example/holodex/playback/data/repository/Media3PlaybackRepositoryImpl.kt
package com.example.holodex.playback.data.repository

import androidx.annotation.OptIn
import androidx.media3.common.MediaItem
import androidx.media3.common.Player
import androidx.media3.common.util.UnstableApi
import com.example.holodex.data.repository.DownloadRepository
import com.example.holodex.data.repository.HolodexRepository
import com.example.holodex.data.repository.UserPreferencesRepository
import com.example.holodex.playback.data.mapper.MediaItemMapper
import com.example.holodex.playback.data.persistence.PlaybackStatePersistenceManager
import com.example.holodex.playback.data.queue.PlaybackQueueManager
import com.example.holodex.playback.data.queue.PlaybackQueueState
import com.example.holodex.playback.data.queue.QueueAction
import com.example.holodex.playback.data.source.StreamResolutionCoordinator
import com.example.holodex.playback.data.tracker.PlaybackProgressTracker
import com.example.holodex.playback.domain.model.DomainPlaybackProgress
import com.example.holodex.playback.domain.model.DomainPlaybackState
import com.example.holodex.playback.domain.model.DomainRepeatMode
import com.example.holodex.playback.domain.model.DomainShuffleMode
import com.example.holodex.playback.domain.model.PersistedPlaybackData
import com.example.holodex.playback.domain.model.PlaybackItem
import com.example.holodex.playback.domain.model.PlaybackQueue
import com.example.holodex.playback.domain.repository.PlaybackRepository
import com.example.holodex.playback.player.Media3PlayerController
import com.example.holodex.playback.util.PlayerStateMapper
import com.example.holodex.playback.util.discontinuityReasonToString
import com.example.holodex.viewmodel.autoplay.ContinuationManager
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.Job
import kotlinx.coroutines.cancel
import kotlinx.coroutines.delay
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.collectLatest
import kotlinx.coroutines.flow.distinctUntilChanged
import kotlinx.coroutines.flow.first
import kotlinx.coroutines.flow.launchIn
import kotlinx.coroutines.flow.map
import kotlinx.coroutines.flow.onEach
```

```
import kotlinx.coroutines.isActive
import kotlinx.coroutines.launch
import kotlinx.coroutines.withContext
import timber.log.Timber

private data class Move(val from: Int, val to: Int)
class PlayerSyncException(message: String, cause: Throwable?) : Exception(message, cause)

@OptIn(UnstableApi::class)
class Media3PlaybackRepositoryImpl(
    private val playerController: Media3PlayerController,
    private val queueManager: PlaybackQueueManager,
    private val streamResolver: StreamResolutionCoordinator,
    private val progressTracker: PlaybackProgressTracker,
    private val persistenceManager: PlaybackStatePersistenceManager,
    private val continuationManager: ContinuationManager,
    private val mediaItemMapper: MediaItemMapper,
    private val downloadRepository: DownloadRepository,
    private val holodexRepository: HolodexRepository,
    private val userPreferencesRepository: UserPreferencesRepository,
    private val repositoryScope: CoroutineScope
) : PlaybackRepository {

    companion object {
        private const val TAG = "M3PlaybackRepo"
        private const val SAVE_DEBOUNCE_MS = 750L
    }

    private var saveStateJob: Job? = null
    override fun observePlaybackState(): Flow<DomainPlaybackState> =
        playerController.playerPlaybackStateFlow

    override fun observePlaybackProgress(): Flow<DomainPlaybackProgress> =
        progressTracker.progressFlow

    override fun observePlaybackQueue(): Flow<PlaybackQueue> =
        queueManager.playbackQueueFlow.map { state ->
            PlaybackQueue(
                queueId = "default_queue",
                items = state.activeList,
                currentIndex = state.currentIndex,
                repeatMode = state.repeatMode,
                shuffleMode = state.shuffleMode
            )
        }.distinctUntilChanged()

    override fun observeCurrentPlayingItem(): Flow<PlaybackItem?> =
        queueManager.playbackQueueFlow.map { it.currentItem }.distinctUntilChanged()


    init {
        Timber.d("$TAG: Initializing...")
        playerController.exoPlayer.playWhenReady = false
        setupPlayerEventListeners()
        setupStateSynchronization()
```

```
    repositoryScope.launch {
        loadInitialState()
        downloadRepository.downloadCompletedEvents.collectLatest { event ->
            handleDownloadCompletion(event.itemId, event.localFileUri)
        }
    }
}

private fun setupStateSynchronization() {
    repositoryScope.launch(Dispatchers.Main.immediate) {
        queueManager.playbackQueueFlow.collect { queueState ->
            // This collector now ONLY handles index and repeat mode changes.
            // Timeline modifications are handled by direct action methods.

            // SYNC 1: Is the player's current track different from our desired track?
            if (playerController.exoPlayer.currentMediaItemIndex != queueState.currentInde
                Timber.d("$TAG Sync: Player index is out of sync. Seeking to ${queueState.
                if (queueState.currentIndex in queueState.activeList.indices) {
                    playerController.seekToItem(queueState.currentIndex, 0L)
                }
            }

            // SYNC 2: Is the player's repeat mode out of sync?
            val newPlayerRepeatMode =
                PlayerStateMapper.mapDomainRepeatModeToExoPlayer(queueState.repeatMode)
            if (playerController.exoPlayer.repeatMode != newPlayerRepeatMode) {
                Timber.d("$TAG Sync: Updating player repeat mode.")
                playerController.setRepeatMode(newPlayerRepeatMode)
            }
        }
    }
}

private fun resolvePlaceholdersInBackground(
    activeList: List<PlaybackItem>,
    alreadyResolvedIndex: Int
) {
    repositoryScope.launch(Dispatchers.IO) {
        activeList.forEachIndexed { index, playbackItem ->
            if (index != alreadyResolvedIndex && isActive) {
                val resolvedItem = streamResolver.resolveSingleStream(playbackItem)
                if (resolvedItem != null) {
                    val finalMediaItem = mediaItemMapper.toMedia3MediaItem(resolvedItem)
                    if (finalMediaItem != null) {
                        withContext(Dispatchers.Main) {
                            val currentQueue = queueManager.playbackQueueFlow.value.active
                            val playerIndex =
                                currentQueue.indexOfFirst { it.id == playbackItem.id }
                            if (playerIndex != -1 && playerIndex < playerController.exoPla
                                playerController.exoPlayer.replaceMediaItem(
                                    playerIndex,
                                    finalMediaItem
                                )
                            }
                        }
```

```
                    }
                } else {
                    val currentQueue = queueManager.playbackQueueFlow.value.activeList
                    val indexInCurrentQueue =
                        currentQueue.indexOfFirst { it.id == playbackItem.id }
                    if (indexInCurrentQueue != -1) {
                        queueManager.dispatch(QueueAction.RemoveItem(indexInCurrentQueue))
                    }
                }
            }
        }
    }
}

override suspend fun prepareAndPlay(
    items: List<PlaybackItem>,
    startIndex: Int,
    startPositionMs: Long,
    shouldShuffle: Boolean
) {
    // --- START OF FIX: Cancel any lingering save job from a previous session ---
    saveStateJob?.cancel()
    // --- END OF FIX ---

    continuationManager.endCurrentSession()
    val startWithShuffle =
        if (shouldShuffle) true else userPreferencesRepository.shuffleOnPlayStartEnabled.f
    setQueueAndPreparePlayer(items, startIndex, startPositionMs, startWithShuffle)
}

override suspend fun prepareAndPlayRadio(radioId: String) {
    // --- START OF FIX: Cancel any lingering save job from a previous session ---
    saveStateJob?.cancel()
    // --- END OF FIX ---

    Timber.tag(TAG).i("RADIO_LOG: prepareAndPlayRadio called with ID: $radioId")
    val initialItems = continuationManager.startRadioSession(radioId, repositoryScope, thi

    if (initialItems.isNullOrEmpty()) {
        Timber.tag(TAG)
            .e("RADIO_LOG: Could not start radio session for $radioId, initial batch was e
        continuationManager.endCurrentSession()
        return
    }

    setQueueAndPreparePlayer(initialItems, 0, 0L, false)
}

private suspend fun setQueueAndPreparePlayer(
    items: List<PlaybackItem>,
    startIndex: Int,
    startPositionMs: Long,
    shouldShuffle: Boolean
) {
    playerController.stop()
```

```
    queueManager.dispatch(
        QueueAction.SetQueue(items, startIndex, startPositionMs, shouldShuffle)
    )

    val finalQueueState = queueManager.playbackQueueFlow.value
    setPlayerTimeline(
        newActiveList = finalQueueState.activeList,
        newCurrentIndex = finalQueueState.currentIndex,
        seekPosition = finalQueueState.transientStartPositionMs
    )
    playerController.play()
}

private fun setPlayerTimeline(
    newActiveList: List<PlaybackItem>,
    newCurrentIndex: Int,
    seekPosition: Long
) {
    if (newActiveList.isEmpty()) {
        repositoryScope.launch(Dispatchers.Main) {
            playerController.clearMediaItemsAndStop()
        }
        return
    }

    val firstItemToPlay = newActiveList.getOrNull(newCurrentIndex)
    if (firstItemToPlay == null) {
        Timber.e("$TAG: setPlayerTimeline failed, invalid start index $newCurrentIndex for
        repositoryScope.launch(Dispatchers.Main) {
            playerController.clearMediaItemsAndStop()
        }
        return
    }

    repositoryScope.launch {
        val resolvedFirstItem = streamResolver.resolveSingleStream(firstItemToPlay)

        withContext(Dispatchers.Main) {
            if (resolvedFirstItem != null) {
                val mediaItems = newActiveList.map {
                    if (it.id == resolvedFirstItem.id) {
                        mediaItemMapper.toMedia3MediaItem(resolvedFirstItem)!!
                    } else {
                        mediaItemMapper.toPlaceholderMediaItem(it)
                    }
                }
                playerController.setMediaItems(mediaItems, newCurrentIndex, seekPosition)
                resolvePlaceholdersInBackground(newActiveList, newCurrentIndex)
            } else {
                Timber.e("$TAG: Failed to resolve the first item to play. Cannot set timel
                playerController.clearMediaItemsAndStop()
            }
        }
    }
```

```
}

override suspend fun play() = playerController.play()
override suspend fun pause() = playerController.pause()
override suspend fun seekTo(positionSec: Long) = playerController.seekTo(positionSec * 100

override suspend fun skipToNext() {
    val currentState = queueManager.playbackQueueFlow.value
    if (currentState.repeatMode == DomainRepeatMode.ONE) {
        playerController.seekTo(0)
    } else {
        queueManager.dispatch(QueueAction.SkipToNext)
    }
}

override suspend fun skipToPrevious() {
    if (playerController.exoPlayer.currentPosition > 3000) {
        playerController.seekTo(0L)
        return
    }
    queueManager.dispatch(QueueAction.SkipToPrevious)
}

override suspend fun skipToQueueItem(index: Int) {
    if (index in queueManager.playbackQueueFlow.value.activeList.indices) {
        queueManager.dispatch(QueueAction.SetCurrentIndex(index))
    }
}

override suspend fun setRepeatMode(mode: DomainRepeatMode) {
    queueManager.dispatch(QueueAction.SetRepeatMode(mode))
}

override suspend fun setShuffleMode(mode: DomainShuffleMode) =
    withContext(Dispatchers.Main.immediate) {
        val currentState = queueManager.playbackQueueFlow.value
        if (mode == currentState.shuffleMode) return@withContext

        try {
            // 1. Update domain state first (Single Source of Truth)
            Timber.d("Toggling shuffle. Current mode: ${currentState.shuffleMode}")
            queueManager.dispatch(QueueAction.ToggleShuffle)
            val newQueueState = queueManager.playbackQueueFlow.value
            Timber.d("Domain state updated. New mode: ${newQueueState.shuffleMode}")

            // 2. Sync player state with the new domain state
            syncPlayerWithQueueState(newQueueState)

        } catch (e: Exception) {
            Timber.e(e, "Failed to sync player for shuffle mode change. Rolling back.")
            // Rollback domain state if player sync fails to maintain consistency
            queueManager.dispatch(QueueAction.ToggleShuffle) // Revert the shuffle
            throw PlayerSyncException("Failed to update shuffle mode", e)
        }
    }
```

```
/**
 * Synchronizes the ExoPlayer's state (timeline order and shuffle mode setting)
 * to match the provided definitive queue state from the PlaybackQueueManager.
 */
private suspend fun syncPlayerWithQueueState(queueState: PlaybackQueueState) {
    // Update the player's shuffle mode setting first
    playerController.exoPlayer.shuffleModeEnabled =
        (queueState.shuffleMode == DomainShuffleMode.ON)

    // Get the current state of the player's timeline
    val currentTimelineIds =
        playerController.getMediaItemsFromPlayerTimeline().map { it.mediaId }
    val desiredOrderIds = queueState.activeList.map { it.id }

    // Only proceed if a reorder is actually necessary
    if (currentTimelineIds == desiredOrderIds) {
        Timber.d("Player timeline already matches desired order. No moves needed.")
        return
    }

    // Calculate the required moves efficiently
    val moves = calculateOptimalMoves(
        current = currentTimelineIds,
        desired = desiredOrderIds
    )

    // Apply the calculated moves to the player
    if (moves.isNotEmpty()) {
        Timber.d("Applying ${moves.size} moves to sync player timeline.")
        applyTimelineChanges(moves)
    }
}

/**
 * Calculates the minimal set of moves needed to transform the current list order
 * into the desired list order. This is more robust than a simple loop.
 */
private fun calculateOptimalMoves(current: List<String>, desired: List<String>): List<Move
    if (current.size != desired.size) {
        // This indicates a severe state inconsistency that should be logged.
        Timber.e("Timeline size mismatch! Current: ${current.size}, Desired: ${desired.siz
        // Returning empty list to prevent crash, but this signals a deeper issue.
        return emptyList()
    }

    val moves = mutableListOf<Move>()
    val workingOrder = current.toMutableList()

    // For each position in the desired final list...
    for (targetIndex in desired.indices) {
        val targetId = desired[targetIndex]
        // ...find where that item currently is in our working copy of the timeline.
        val currentIndex = workingOrder.indexOf(targetId)
```

```
        // If the item is not where it's supposed to be...
        if (currentIndex != targetIndex && currentIndex != -1) {
            // ...record the move we need to make.
            moves.add(Move(from = currentIndex, to = targetIndex))
            // And simulate that move in our working copy so the next iteration's
            // `indexOf` call is accurate.
            workingOrder.add(targetIndex, workingOrder.removeAt(currentIndex))
        }
    }

    return moves
}

/**
 * Applies a list of move operations to the player's timeline.
 * Currently, this is done sequentially as Media3 does not have a batch-move command.
 */
private suspend fun applyTimelineChanges(moves: List<Move>) {
    // ExoPlayer processes commands on its own thread sequentially.
    // Sending them one after another is the correct approach.
    moves.forEach { move ->
        playerController.exoPlayer.moveMediaItem(move.from, move.to)
    }
}

override suspend fun reorderQueueItem(fromIndex: Int, toIndex: Int) {
    // --- START OF REFACTORED REORDER LOGIC ---
    withContext(Dispatchers.Main.immediate) {
        // Directly command the player
        playerController.exoPlayer.moveMediaItem(fromIndex, toIndex)
        // Dispatch to our SSoT to keep it in sync
        queueManager.dispatch(QueueAction.ReorderItem(fromIndex, toIndex))
    }
    // --- END OF REFACTORED REORDER LOGIC ---
}

override suspend fun addItemToQueue(item: PlaybackItem, index: Int?) {
    // --- START OF REFACTORED ADD LOGIC ---
    withContext(Dispatchers.Main.immediate) {
        // First, determine the final insertion index from our SSoT's rules
        val currentState = queueManager.playbackQueueFlow.value
        val finalIndex =
            queueManager.calculateInsertionIndex(item, index) // <-- We need to add this h

        // Directly command the player
        val mediaItem = mediaItemMapper.toPlaceholderMediaItem(item)
        playerController.exoPlayer.addMediaItem(finalIndex, mediaItem)

        // Dispatch to our SSoT to keep it in sync
        queueManager.dispatch(QueueAction.AddItem(item, index))

        // Resolve the new placeholder in the background
        resolvePlaceholdersInBackground(listOf(item), -1)
    }
    // --- END OF REFACTORED ADD LOGIC ---
```

```
}

// The public addItemsToQueue method remains UNCHANGED. It's for UI-initiated actions.
override suspend fun addItemsToQueue(items: List<PlaybackItem>, index: Int?) {
    withContext(Dispatchers.Main.immediate) {
        val currentState = queueManager.playbackQueueFlow.value
        val finalIndex = queueManager.calculateInsertionIndex(items.first(), index)

        // This correctly creates placeholders for a responsive UI
        val mediaItems = items.map { mediaItemMapper.toPlaceholderMediaItem(it) }
        playerController.exoPlayer.addMediaItems(finalIndex, mediaItems)

        queueManager.dispatch(QueueAction.AddItems(items, index))

        resolvePlaceholdersInBackground(items, -1)
    }
}

// --- NEW: A private helper for adding pre-made MediaItems ---
/**
 * A private helper to add already-created MediaItems to the player and update the queue.
 * This bypasses the placeholder creation logic and is used for internal operations
 * like autoplay where items are pre-resolved.
 */
private suspend fun addResolvedMediaItemsToQueue(
    playbackItems: List<PlaybackItem>,
    mediaItems: List<MediaItem>
) {
    withContext(Dispatchers.Main.immediate) {
        val currentState = queueManager.playbackQueueFlow.value
        // Autoplay and Radio always append to the end.
        val finalIndex = currentState.activeList.size

        playerController.exoPlayer.addMediaItems(finalIndex, mediaItems)
        queueManager.dispatch(QueueAction.AddItems(playbackItems, null))
    }
}

override suspend fun removeItemFromQueue(index: Int) {
    // --- START OF REFACTORED REMOVE LOGIC ---
    withContext(Dispatchers.Main.immediate) {
        // Directly command the player
        playerController.exoPlayer.removeMediaItem(index)
        // Dispatch to our SSoT to keep it in sync
        queueManager.dispatch(QueueAction.RemoveItem(index))
    }
    // --- END OF REFACTORED REMOVE LOGIC ---
}

override suspend fun clearQueue() {
    withContext(Dispatchers.Main.immediate) {
        saveStateJob?.cancel()
        saveStateJob = null

        playerController.clearMediaItemsAndStop()
```

```
        queueManager.dispatch(QueueAction.ClearQueue)
        persistenceManager.clearState()
        Timber.tag(TAG).i("Cleared both live and persisted playback state.")
    }
}


private fun setupPlayerEventListeners() {
    playerController.playerPlaybackStateFlow.onEach { state ->
        if (state == DomainPlaybackState.ENDED) handlePlaybackEndedByPlayer()
        // --- START OF FIX: Only schedule a save on legitimate state changes, not BUFFERI
        if (state != DomainPlaybackState.BUFFERING) {
            scheduleSaveState()
        }
        // --- END OF FIX ---
    }.launchIn(repositoryScope)

    playerController.isPlayingChangedEventFlow.onEach { event ->
        if (event.isPlaying) progressTracker.startTracking() else progressTracker.stopTrac
    }.launchIn(repositoryScope)

    playerController.mediaItemTransitionEventFlow.onEach { event ->
        when (event.reason) {
            Player.MEDIA_ITEM_TRANSITION_REASON_SEEK -> progressTracker.resetProgress()
            Player.MEDIA_ITEM_TRANSITION_REASON_AUTO -> {
                handleAutoTransition(event.newIndex)
                progressTracker.resetProgress()
            }

            else -> progressTracker.resetProgress()
        }
    }.launchIn(repositoryScope)

    playerController.discontinuityEventFlow.onEach { event ->
        if (event.reason == Player.DISCONTINUITY_REASON_SKIP || event.reason == Player.DIS
            Timber.d("Player discontinuity event. Reason: ${discontinuityReasonToString(ev
        }
    }.launchIn(repositoryScope)
}

private fun handleAutoTransition(playerIndex: Int) {
    if (playerIndex != queueManager.playbackQueueFlow.value.currentIndex) {
        queueManager.dispatch(QueueAction.SetCurrentIndex(playerIndex))
    }
}

private fun handleDownloadCompletion(itemId: String, localFileUri: String) {
    val itemToUpdate =
        queueManager.playbackQueueFlow.value.activeList.find { it.id == itemId } ?: return
    val updatedItem = itemToUpdate.copy(streamUri = localFileUri)
    queueManager.dispatch(QueueAction.UpdateItemInQueue(updatedItem))
}

private suspend fun loadInitialState() {
    val persistedData = persistenceManager.loadState() ?: return
```

```
    val domainItems = persistedData.queueItems.mapNotNull { mediaItemMapper.toPlaybackItem
    val restoredShuffledDomainItems = if (persistedData.shuffleMode == DomainShuffleMode.O
        persistedData.shuffledQueueItemIds?.mapNotNull { id -> domainItems.find { it.id ==
    } else {
        null
    }

    // 1. Dispatch to the QueueManager to set our app's internal state.
    queueManager.dispatch(
        QueueAction.SetQueue(
            domainItems,
            persistedData.currentIndex,
            persistedData.currentPositionSec * 1000L,
            false, // Don't re-shuffle on restore
            persistedData.shuffleMode,
            persistedData.repeatMode,
            restoredShuffledDomainItems
        )
    )

    // --- FIX: Add this block to synchronize the player with the newly loaded state ---
    // 2. Get the definitive state that was just set in the QueueManager.
    val finalQueueState = queueManager.playbackQueueFlow.value

    // 3. Build the player's timeline based on this restored state.
    //    We do NOT automatically start playback here; we just prepare the player.
    if (finalQueueState.activeList.isNotEmpty()) {
        Timber.d("$TAG: Restoring player timeline with ${finalQueueState.activeList.size}
        setPlayerTimeline(
            newActiveList = finalQueueState.activeList,
            newCurrentIndex = finalQueueState.currentIndex,
            seekPosition = finalQueueState.transientStartPositionMs
        )
    }
    // --- END OF FIX ---
}

private suspend fun generatePersistedPlaybackData(): PersistedPlaybackData? {
    val currentQueueState = queueManager.playbackQueueFlow.value
    if (currentQueueState.originalList.isEmpty()) return null
    return withContext(Dispatchers.Main.immediate) {
        PersistedPlaybackData(
            "default_queue",
            currentQueueState.originalList.map { mediaItemMapper.toPersistedPlaybackItem(i
            playerController.exoPlayer.currentMediaItemIndex,
            playerController.exoPlayer.currentPosition / 1000L,
            currentQueueState.currentItem?.id,
            currentQueueState.repeatMode,
            currentQueueState.shuffleMode,
            if (currentQueueState.shuffleMode == DomainShuffleMode.ON) currentQueueState.a
        )
    }
}

private fun scheduleSaveState() {
```

```
    // --- START OF FIX: Implement debouncing with job cancellation ---
    saveStateJob?.cancel()
    saveStateJob = repositoryScope.launch {
        delay(SAVE_DEBOUNCE_MS)
        generatePersistedPlaybackData()?.let {
            persistenceManager.saveState(it) // Use direct save, not scheduled
        }
    }
    // --- END OF FIX ---
}

private suspend fun handlePlaybackEndedByPlayer() {
    withContext(Dispatchers.Main.immediate) {
        val currentQueueState = queueManager.playbackQueueFlow.value
        val autoplayItems =
            continuationManager.provideAutoplayItems(currentQueueState.activeList)

        if (!autoplayItems.isNullOrEmpty()) {
            Timber.i("$TAG: Autoplay provided ${autoplayItems.size} new items. Resolving f

            val firstItemToPlay = autoplayItems.first()
            val resolvedFirstItem = streamResolver.resolveSingleStream(firstItemToPlay)

            if (resolvedFirstItem == null) {
                Timber.e("$TAG: Failed to resolve stream for the first autoplay item. Abor
                playerController.pause()
                return@withContext
            }

            // 1. Create the list of MediaItems to be added. One real, the rest placeholde
            val newMediaItems = mutableListOf<MediaItem>()
            newMediaItems.add(mediaItemMapper.toMedia3MediaItem(resolvedFirstItem)!!)

            if (autoplayItems.size > 1) {
                autoplayItems.subList(1, autoplayItems.size).forEach { item ->
                    newMediaItems.add(mediaItemMapper.toPlaceholderMediaItem(item))
                }
            }

            // 2. Use our new private helper to add the items without re-creating placehol
            addResolvedMediaItemsToQueue(autoplayItems, newMediaItems)

            skipToNext()

            if (autoplayItems.size > 1) {
                resolvePlaceholdersInBackground(
                    autoplayItems.subList(1, autoplayItems.size),
                    -1
                )
            }

        } else {
            Timber.i("$TAG: Playback ended and no autoplay items were provided. Pausing.")
            playerController.pause()
        }
```

```
        }
    }


    override suspend fun setScrubbing(isScrubbing: Boolean) {
        playerController.exoPlayer.setScrubbingModeEnabled(isScrubbing)
    }

    override fun release() {
        repositoryScope.cancel()
        progressTracker.stopTracking()
        playerController.releasePlayer()
    }

    override fun getPlayerSessionId(): Int? =
        playerController.exoPlayer.audioSessionId.takeIf { it != -1 }
}



// File: java\com\example\holodex\playback\data\repository\RoomPlaybackStateRepositoryImpl.kt
// File: java/com/example/holodex/playback/data/repository/RoomPlaybackStateRepositoryImpl.kt
package com.example.holodex.playback.data.repository

import com.example.holodex.playback.data.mapper.toDomainModel
import com.example.holodex.playback.data.mapper.toEntities
import com.example.holodex.playback.data.model.PersistedPlaybackStateDao
import com.example.holodex.playback.domain.model.PersistedPlaybackData
import com.example.holodex.playback.domain.repository.PlaybackStateRepository

class RoomPlaybackStateRepositoryImpl(
    private val dao: PersistedPlaybackStateDao
) : PlaybackStateRepository {

    private val defaultQueueId = "default_queue"

    override suspend fun saveState(data: PersistedPlaybackData) {
        val (stateEntity, itemEntities) = data.toEntities()
        dao.savePlaybackStateWithItems(stateEntity, itemEntities)
    }

    override suspend fun loadState(): PersistedPlaybackData? {
        val stateWithItems = dao.getPlaybackStateWithItems(defaultQueueId)
        return stateWithItems?.let { tuple ->
            tuple.state.toDomainModel(tuple.items.sortedBy { it.itemOrder })
        }
    }

    override suspend fun clearState() {
        dao.clearPlaybackStateByQueueId(defaultQueueId)
        dao.clearPlaybackItemsByQueueId(defaultQueueId)
    }
}

// File: java\com\example\holodex\playback\data\source\HolodexResolvingDataSource.kt
```

```
package com.example.holodex.playback.data.source

import android.net.Uri
import androidx.media3.common.util.UnstableApi
import androidx.media3.datasource.DataSpec
import androidx.media3.datasource.ResolvingDataSource
import com.example.holodex.data.repository.YouTubeStreamRepository
import kotlinx.coroutines.runBlocking
import timber.log.Timber
import javax.inject.Inject

/**
 * Intercepts "holodex://" URIs.
 * Resolves the actual audio stream URL (m4a/webm) just-in-time.
 * This allows the UI to set the queue instantly without waiting for network calls.
 */
@UnstableApi
class HolodexResolvingDataSource @Inject constructor(
    private val streamRepository: YouTubeStreamRepository
) : ResolvingDataSource.Resolver {

    override fun resolveDataSpec(dataSpec: DataSpec): DataSpec {
        val uri = dataSpec.uri

        // We only intercept our custom scheme
        if (uri.scheme == "holodex" && uri.host == "resolve") {
            val videoId = uri.lastPathSegment ?: return dataSpec

            Timber.d("ResolvingDataSource: JIT Resolving for $videoId")

            // runBlocking is necessary here because this API is synchronous.
            // It runs on ExoPlayer's background loading thread, so it DOES NOT block the UI.
            val streamDetails = try {
                runBlocking {
                    streamRepository.getAudioStreamDetails(videoId).getOrNull()
                }
            } catch (e: Exception) {
                Timber.e(e, "Failed to resolve stream for $videoId")
                null
            }

            if (streamDetails != null) {
                Timber.i("ResolvingDataSource: Resolved $videoId -> ${streamDetails.streamUrl}
                return dataSpec.buildUpon()
                    .setUri(Uri.parse(streamDetails.streamUrl))
                    .build()
            } else {
                Timber.e("ResolvingDataSource: Failed to resolve URL for $videoId")
                // We throw here to trigger the Player's error handling state
                throw java.io.IOException("Could not resolve stream for video $videoId")
            }
        }

        // Pass through local files and standard HTTP urls
        return dataSpec
```

```
    }
}

// File: java\com\example\holodex\playback\data\source\StreamResolutionCoordinator.kt
// File: java/com/example/holodex/playback/data/source/StreamResolutionCoordinator.kt
package com.example.holodex.playback.data.source

import androidx.collection.LruCache
import androidx.media3.common.util.UnstableApi
import com.example.holodex.data.db.DownloadStatus
import com.example.holodex.data.db.DownloadedItemDao
import com.example.holodex.data.db.DownloadedItemEntity
import com.example.holodex.playback.domain.model.PlaybackItem
import com.example.holodex.playback.domain.model.StreamDetails
import com.example.holodex.playback.domain.repository.StreamResolverRepository
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.Job
import kotlinx.coroutines.isActive
import kotlinx.coroutines.withContext
import java.util.concurrent.TimeUnit
import kotlin.coroutines.coroutineContext

@UnstableApi
class StreamResolutionCoordinator(
    private val streamResolverRepository: StreamResolverRepository,
    private val downloadedItemDao: DownloadedItemDao
) {
    companion object {
        private const val TAG = "StreamResolutionCoord"
    }

    private var currentResolutionJob: Job? = null
    private data class ResolvedStreamCacheEntry(val streamDetails: StreamDetails, val timestam
    private val streamUrlCache = LruCache<String, ResolvedStreamCacheEntry>(50)
    private val cacheExpiryMs = TimeUnit.MINUTES.toMillis(10)

    suspend fun resolveSingleStream(item: PlaybackItem): PlaybackItem? {
        return withContext(Dispatchers.IO) {
            resolveSingleStreamInternal(item.copy(), null)
        }
    }

    fun clearMemoryCache() {
        streamUrlCache.evictAll()
    }

    private suspend fun resolveSingleStreamInternal(
        item: PlaybackItem,
        prewarmedDownloads: Map<String, DownloadedItemEntity>?
    ): PlaybackItem? {
        if (!item.streamUri.isNullOrBlank()) {
            return item
        }

        val downloadedItem = prewarmedDownloads?.get(item.id) ?: downloadedItemDao.getById(ite
```

```
        if (downloadedItem != null && downloadedItem.downloadStatus == DownloadStatus.COMPLETE
            return item.copy(streamUri = downloadedItem.localFileUri)
        }

        getStreamFromCache(item.videoId)?.let { cachedDetails ->
            return item.copy(streamUri = cachedDetails.url)
        }

        if (!coroutineContext.isActive) {
            return null
        }

        return try {
            val result = streamResolverRepository.resolveStreamUrl(item.videoId)
            if (result.isSuccess) {
                val streamDetails = result.getOrThrow()
                putStreamInCache(item.videoId, streamDetails)
                item.copy(streamUri = streamDetails.url)
            } else {
                null
            }
        } catch (e: Exception) {
            null
        }
    }

    private fun getStreamFromCache(videoId: String): StreamDetails? {
        val entry = streamUrlCache[videoId]
        if (entry != null) {
            if (System.currentTimeMillis() - entry.timestamp < cacheExpiryMs) {
                return entry.streamDetails
            } else {
                streamUrlCache.remove(videoId)
            }
        }
        return null
    }

    private fun putStreamInCache(videoId: String, streamDetails: StreamDetails) {
        streamUrlCache.put(videoId, ResolvedStreamCacheEntry(streamDetails, System.currentTime
    }

    fun cancelOngoingResolutions() {
        synchronized(this) {
            currentResolutionJob?.cancel()
            currentResolutionJob = null
        }
    }
}

// File: java\com\example\holodex\playback\data\tracker\PlaybackProgressTracker.kt
// File: java/com/example/holodex/playback/data/tracker/PlaybackProgressTracker.kt
package com.example.holodex.playback.data.tracker

import androidx.media3.common.C
```

```
import androidx.media3.common.Player
import com.example.holodex.data.repository.HolodexRepository
import com.example.holodex.playback.data.mapper.MediaItemMapper
import com.example.holodex.playback.domain.model.DomainPlaybackProgress
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.Job
import kotlinx.coroutines.delay
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.isActive
import kotlinx.coroutines.launch
import kotlinx.coroutines.withContext
import timber.log.Timber

class PlaybackProgressTracker(
    private val exoPlayer: Player,
    private val scope: CoroutineScope,
    private val holodexRepository: HolodexRepository,
    private val mediaItemMapper: MediaItemMapper
) {
    companion object {
        private const val TAG = "PlaybackProgressTracker"
        private const val HISTORY_SAVE_THRESHOLD_PERCENT = 50.0
    }

    private val _progressFlow = MutableStateFlow(DomainPlaybackProgress.NONE)
    val progressFlow: StateFlow<DomainPlaybackProgress> = _progressFlow.asStateFlow()

    private var progressUpdateJob: Job? = null
    private var currentMediaIdForHistory: String? = null
    private var hasBeenSavedToHistory: Boolean = false

    fun startTracking() {
        Timber.d("$TAG: Start tracking progress.")
        stopTracking()
        scope.launch {
            if (withContext(Dispatchers.Main) { exoPlayer.isPlaying }) {
                progressUpdateJob = launch {
                    while (isActive) {
                        updateProgress()
                        delay(1000L)
                    }
                }
            }
        }
    }

    fun stopTracking() {
        progressUpdateJob?.cancel()
        progressUpdateJob = null
    }

    private suspend fun updateProgress() {
```

```
        if (!scope.isActive) return

        withContext(Dispatchers.Main) {
            val currentMediaItem = exoPlayer.currentMediaItem
            val currentMediaId = currentMediaItem?.mediaId
            val currentPositionMs = exoPlayer.currentPosition.coerceAtLeast(0L)
            val playerDurationMs = exoPlayer.duration.takeIf { it != C.TIME_UNSET } ?: 0L

            val newProgress = DomainPlaybackProgress(
                positionSec = currentPositionMs / 1000L,
                durationSec = playerDurationMs / 1000L,
                bufferedPositionSec = exoPlayer.bufferedPosition.coerceAtLeast(0L) / 1000L
            )
            if (_progressFlow.value != newProgress) {
                _progressFlow.value = newProgress
            }

            if (currentMediaId == null) {
                if (currentMediaIdForHistory != null) {
                    currentMediaIdForHistory = null
                    hasBeenSavedToHistory = false
                }
                return@withContext
            }

            if (currentMediaId != currentMediaIdForHistory) {
                currentMediaIdForHistory = currentMediaId
                hasBeenSavedToHistory = false
            }

            if (hasBeenSavedToHistory) return@withContext

            val playbackPercentage = (currentPositionMs.toDouble() / playerDurationMs.toDouble
            val isEligible = playbackPercentage >= HISTORY_SAVE_THRESHOLD_PERCENT

            val playbackItem = currentMediaItem.let { mediaItemMapper.toPlaybackItem(it) }
            val hasResolvedStream = !playbackItem?.streamUri.isNullOrBlank()

            if (isEligible && hasResolvedStream) {
                holodexRepository.addSongToHistory(playbackItem!!)
                hasBeenSavedToHistory = true
            }
        }
    }

    fun resetProgress() {
        _progressFlow.value = DomainPlaybackProgress.NONE
    }
}

// File: java\com\example\holodex\playback\domain\model\DomainPlaybackProgress.kt
// File: java/com/example/holodex/playback/domain/model/DomainPlaybackProgress.kt
package com.example.holodex.playback.domain.model

data class DomainPlaybackProgress(
```

```
    val positionSec: Long = 0L,
    val durationSec: Long = 0L,
    val bufferedPositionSec: Long = 0L
) {
    companion object {
        val NONE = DomainPlaybackProgress()
    }
}

// File: java\com\example\holodex\playback\domain\model\DomainPlaybackState.kt
// File: java/com/example/holodex/playback/domain/model/DomainPlaybackState.kt
package com.example.holodex.playback.domain.model

enum class DomainPlaybackState {
    IDLE,
    BUFFERING,
    PLAYING,
    PAUSED,
    ENDED,
    ERROR
}

// File: java\com\example\holodex\playback\domain\model\DomainRepeatMode.kt
// File: java/com/example/holodex/playback/domain/model/DomainRepeatMode.kt
package com.example.holodex.playback.domain.model

enum class DomainRepeatMode {
    NONE,
    ONE,
    ALL
}

// File: java\com\example\holodex\playback\domain\model\DomainShuffleMode.kt
// File: java/com/example/holodex/playback/domain/model/DomainShuffleMode.kt
package com.example.holodex.playback.domain.model

enum class DomainShuffleMode {
    OFF,
    ON
}

// File: java\com\example\holodex\playback\domain\model\PersistedPlaybackData.kt
// File: java/com/example/holodex/playback/domain/model/PersistedPlaybackData.kt
package com.example.holodex.playback.domain.model

data class PersistedPlaybackData(
    val queueId: String,
    val queueItems: List<PersistedPlaybackItem>,
    val currentIndex: Int,
    val currentPositionSec: Long,
    val currentItemId: String?,
    val repeatMode: DomainRepeatMode,
    val shuffleMode: DomainShuffleMode,
    val shuffledQueueItemIds: List<String>? = null
)
```

```
data class PersistedPlaybackItem(
    val id: String,
    val videoId: String,
    val songId: String?,
    val title: String,
    val artistText: String,
    val albumText: String?,
    val artworkUri: String?,
    val durationSec: Long,
    val description: String? = null,
    val channelId: String,
    val clipStartSec: Long? = null,
    val clipEndSec: Long? = null
)

// File: java\com\example\holodex\playback\domain\model\PlaybackItem.kt
// File: java/com/example/holodex/playback/domain/model/PlaybackItem.kt
package com.example.holodex.playback.domain.model

import android.os.Parcelable
import kotlinx.parcelize.Parcelize

@Parcelize
data class PlaybackItem(
    /** The unique composite ID for this item, used for playback and UI state. (e.g., "videoId
    val id: String,

    /** The ID of the parent YouTube video. */
    val videoId: String,

    /** The server's unique UUID for this specific song segment. Null for full videos. */
    val serverUuid: String?,

    /** DEPRECATED USAGE: This was used ambiguously. Now primarily for internal reference if n
    val songId: String?,

    val title: String,
    val artistText: String,
    val albumText: String?,
    val artworkUri: String?,
    val durationSec: Long,
    var streamUri: String? = null,
    val clipStartSec: Long? = null,
    val clipEndSec: Long? = null,
    val description: String? = null,
    val channelId: String,
    val originalArtist: String? = null,

    // *** THIS IS THE MISSING PROPERTY ***
    val isExternal: Boolean = false
) : Parcelable

// File: java\com\example\holodex\playback\domain\model\PlaybackQueue.kt
// File: java/com/example/holodex/playback/domain/model/PlaybackQueue.kt
```

```
package com.example.holodex.playback.domain.model

data class PlaybackQueue(
    val items: List<PlaybackItem> = emptyList(),
    val currentIndex: Int = -1,
    val repeatMode: DomainRepeatMode = DomainRepeatMode.NONE,
    val shuffleMode: DomainShuffleMode = DomainShuffleMode.OFF,
    val queueId: String = "default_queue"
) {
    val currentItem: PlaybackItem?
        get() = items.getOrNull(currentIndex)
}

// File: java\com\example\holodex\playback\domain\model\StreamDetails.kt
// File: java/com/example/holodex/playback/domain/model/StreamDetails.kt
package com.example.holodex.playback.domain.model

data class StreamDetails(
    val url: String,
    val format: String?,
    val quality: String?
)

// File: java\com\example\holodex\playback\domain\repository\PlaybackRepository.kt
// File: java/com/example/holodex/playback/domain/repository/PlaybackRepository.kt
package com.example.holodex.playback.domain.repository

import com.example.holodex.playback.domain.model.DomainPlaybackProgress
import com.example.holodex.playback.domain.model.DomainPlaybackState
import com.example.holodex.playback.domain.model.DomainRepeatMode
import com.example.holodex.playback.domain.model.DomainShuffleMode
import com.example.holodex.playback.domain.model.PlaybackItem
import com.example.holodex.playback.domain.model.PlaybackQueue
import kotlinx.coroutines.flow.Flow

interface PlaybackRepository {
    suspend fun prepareAndPlay(items: List<PlaybackItem>, startIndex: Int, startPositionMs: Lo
    suspend fun prepareAndPlayRadio(radioId: String)
    suspend fun play()
    suspend fun pause()
    suspend fun seekTo(positionSec: Long) // Changed to Long
    suspend fun skipToNext()
    suspend fun skipToPrevious()
    suspend fun setRepeatMode(mode: DomainRepeatMode)
    suspend fun setShuffleMode(mode: DomainShuffleMode)
    suspend fun addItemToQueue(item: PlaybackItem, index: Int?)
    suspend fun addItemsToQueue(items: List<PlaybackItem>, index: Int? = null)
    suspend fun removeItemFromQueue(index: Int)
    suspend fun reorderQueueItem(fromIndex: Int, toIndex: Int)
    suspend fun clearQueue()
    suspend fun setScrubbing(isScrubbing: Boolean)
    fun observePlaybackState(): Flow<DomainPlaybackState>
    fun observePlaybackProgress(): Flow<DomainPlaybackProgress>
    fun observeCurrentPlayingItem(): Flow<PlaybackItem?>
    fun observePlaybackQueue(): Flow<PlaybackQueue>
```

```
    suspend fun skipToQueueItem(index: Int)
    fun release()
    fun getPlayerSessionId(): Int?
}

// File: java\com\example\holodex\playback\domain\repository\PlaybackStateRepository.kt
// File: java/com/example/holodex/playback/domain/repository/PlaybackStateRepository.kt
package com.example.holodex.playback.domain.repository

import com.example.holodex.playback.domain.model.PersistedPlaybackData

interface PlaybackStateRepository {
    suspend fun saveState(data: PersistedPlaybackData)
    suspend fun loadState(): PersistedPlaybackData?
    suspend fun clearState()
}

// File: java\com\example\holodex\playback\domain\repository\StreamResolverRepository.kt
// File: java/com/example/holodex/playback/domain/repository/StreamResolverRepository.kt
package com.example.holodex.playback.domain.repository

import com.example.holodex.playback.domain.model.StreamDetails

interface StreamResolverRepository {
    suspend fun resolveStreamUrl(videoId: String): Result<StreamDetails>
}

// File: java\com\example\holodex\playback\domain\usecase\AddItemsToQueueUseCase.kt
package com.example.holodex.playback.domain.usecase

import com.example.holodex.playback.domain.model.PlaybackItem
import com.example.holodex.playback.domain.repository.PlaybackRepository


class AddItemsToQueueUseCase(private val playbackRepository: PlaybackRepository) {
    // Appends to the end if index is null
    suspend operator fun invoke(items: List<PlaybackItem>, index: Int? = null) {
        playbackRepository.addItemsToQueue(items, index)
    }
}

// File: java\com\example\holodex\playback\domain\usecase\AddItemToQueueUseCase.kt
// File: java/com/example/holodex/playback/domain/usecase/AddItemToQueueUseCase.kt
package com.example.holodex.playback.domain.usecase

import com.example.holodex.playback.domain.model.PlaybackItem
import com.example.holodex.playback.domain.repository.PlaybackRepository

class AddItemToQueueUseCase(private val playbackRepository: PlaybackRepository) {
    suspend operator fun invoke(item: PlaybackItem, index: Int? = null) {
        playbackRepository.addItemToQueue(item, index)
    }
}

// File: java\com\example\holodex\playback\domain\usecase\AddOrFetchAndAddUseCase.kt
```

```
// File: java/com/example/holodex/playback/domain/usecase/AddOrFetchAndAddUseCase.kt
package com.example.holodex.playback.domain.usecase

import com.example.holodex.data.repository.HolodexRepository
import com.example.holodex.playback.domain.model.PlaybackItem
import com.example.holodex.viewmodel.mappers.toPlaybackItem
import timber.log.Timber
import javax.inject.Inject

/**
 * A use case that intelligently adds items to the playback queue.
 * - If the item is a song segment, it's added directly.
 * - If the item is a full video, it fetches the video's segments and adds all of them.
 * - If a video has no segments, it adds the full video itself.
 * @return A Result containing a user-friendly message for a Toast/Snackbar.
 */
class AddOrFetchAndAddUseCase @Inject constructor(
    private val holodexRepository: HolodexRepository,
    private val addItemsToQueueUseCase: AddItemsToQueueUseCase
) {
    suspend operator fun invoke(item: PlaybackItem): Result<String> {
        return try {
            if (item.songId != null) {
                // It's a single song segment, add it directly.
                addItemsToQueueUseCase(listOf(item))
                Timber.d("AddOrFetchAndAddUseCase: Added single segment '${item.title}' to que
                Result.success("Added '${item.title}' to queue.")
            } else {
                // It's a full video, fetch its details to get all segments.
                Timber.d("AddOrFetchAndAddUseCase: Item is a full video ('${item.title}'). Fet
                val videoResult = holodexRepository.getVideoWithSongs(item.videoId, forceRefre

                videoResult.fold(
                    onSuccess = { videoWithSongs ->
                        if (!videoWithSongs.songs.isNullOrEmpty()) {
                            // Video has segments, add them all.
                            val segmentItems = videoWithSongs.songs.map { song ->
                                song.toPlaybackItem(videoWithSongs)
                            }
                            addItemsToQueueUseCase(segmentItems)
                            Timber.d("AddOrFetchAndAddUseCase: Added ${segmentItems.size} segm
                            Result.success("Added ${segmentItems.size} songs from '${item.titl
                        } else {
                            // Video has no segments, add the full video itself.
                            addItemsToQueueUseCase(listOf(item))
                            Timber.d("AddOrFetchAndAddUseCase: Video has no segments. Added fu
                            Result.success("Added '${item.title}' to queue.")
                        }
                    },
                    onFailure = {
                        Timber.e(it, "AddOrFetchAndAddUseCase: Failed to fetch video details f
                        Result.failure(it)
                    }
                )
            }
```

```
        } catch (e: Exception) {
            Timber.e(e, "AddOrFetchAndAddUseCase: An unexpected error occurred.")
            Result.failure(e)
        }
    }
}

// File: java\com\example\holodex\playback\domain\usecase\ClearQueueUseCase.kt
package com.example.holodex.playback.domain.usecase

import com.example.holodex.playback.domain.repository.PlaybackRepository


class ClearQueueUseCase(private val playbackRepository: PlaybackRepository) {
    suspend operator fun invoke() {
        playbackRepository.clearQueue()
    }
}

// File: java\com\example\holodex\playback\domain\usecase\GetPlayerSessionIdUseCase.kt
package com.example.holodex.playback.domain.usecase

import com.example.holodex.playback.domain.repository.PlaybackRepository


class GetPlayerSessionIdUseCase(
    private val playbackRepository: PlaybackRepository
) {
    operator fun invoke(): Int? = playbackRepository.getPlayerSessionId()
}

// File: java\com\example\holodex\playback\domain\usecase\LoadPlaybackStateUseCase.kt
package com.example.holodex.playback.domain.usecase

import com.example.holodex.playback.domain.model.PersistedPlaybackData
import com.example.holodex.playback.domain.repository.PlaybackStateRepository

class LoadPlaybackStateUseCase(
    private val playbackStateRepository: PlaybackStateRepository
) {
    suspend operator fun invoke(): PersistedPlaybackData? {
        return playbackStateRepository.loadState()
    }
}

// File: java\com\example\holodex\playback\domain\usecase\ObserveCurrentPlayingItemUseCase.kt
package com.example.holodex.playback.domain.usecase

import com.example.holodex.playback.domain.model.PlaybackItem
import com.example.holodex.playback.domain.repository.PlaybackRepository
import kotlinx.coroutines.flow.Flow


class ObserveCurrentPlayingItemUseCase(
    private val playbackRepository: PlaybackRepository
```

```
) {
    operator fun invoke(): Flow<PlaybackItem?> = playbackRepository.observeCurrentPlayingItem(
}

// File: java\com\example\holodex\playback\domain\usecase\ObservePlaybackProgressUseCase.kt
package com.example.holodex.playback.domain.usecase

import com.example.holodex.playback.domain.model.DomainPlaybackProgress
import com.example.holodex.playback.domain.repository.PlaybackRepository
import kotlinx.coroutines.flow.Flow


class ObservePlaybackProgressUseCase(
    private val playbackRepository: PlaybackRepository
) {
    operator fun invoke(): Flow<DomainPlaybackProgress> = playbackRepository.observePlaybackPr
}

// File: java\com\example\holodex\playback\domain\usecase\ObservePlaybackQueueUseCase.kt
package com.example.holodex.playback.domain.usecase

import com.example.holodex.playback.domain.model.PlaybackQueue
import com.example.holodex.playback.domain.repository.PlaybackRepository
import kotlinx.coroutines.flow.Flow


class ObservePlaybackQueueUseCase(
    private val playbackRepository: PlaybackRepository
) {
    operator fun invoke(): Flow<PlaybackQueue> = playbackRepository.observePlaybackQueue()
}

// File: java\com\example\holodex\playback\domain\usecase\ObservePlaybackStateUseCase.kt
package com.example.holodex.playback.domain.usecase

import com.example.holodex.playback.domain.model.DomainPlaybackState
import com.example.holodex.playback.domain.repository.PlaybackRepository
import kotlinx.coroutines.flow.Flow

class ObservePlaybackStateUseCase(
    private val playbackRepository: PlaybackRepository
) {
    operator fun invoke(): Flow<DomainPlaybackState> = playbackRepository.observePlaybackState
}

// File: java\com\example\holodex\playback\domain\usecase\PausePlaybackUseCase.kt
package com.example.holodex.playback.domain.usecase

import com.example.holodex.playback.domain.repository.PlaybackRepository


class PausePlaybackUseCase(private val playbackRepository: PlaybackRepository) {
    suspend operator fun invoke() {
        playbackRepository.pause()
    }
```

```
}

// File: java\com\example\holodex\playback\domain\usecase\PlayerControlUseCase.kt
package com.example.holodex.playback.domain.usecase

import com.example.holodex.playback.domain.model.DomainRepeatMode
import com.example.holodex.playback.domain.model.DomainShuffleMode
import com.example.holodex.playback.domain.model.PlaybackItem
import com.example.holodex.playback.domain.repository.PlaybackRepository
import javax.inject.Inject

class PlayerControlUseCase @Inject constructor(
    private val repository: PlaybackRepository
) {
    suspend fun play() = repository.play()
    suspend fun pause() = repository.pause()
    suspend fun resume() = repository.play()
    suspend fun seekTo(positionSec: Long) = repository.seekTo(positionSec)
    suspend fun skipToNext() = repository.skipToNext()
    suspend fun skipToPrevious() = repository.skipToPrevious()
    suspend fun skipToQueueIndex(index: Int) = repository.skipToQueueItem(index)

    suspend fun setRepeatMode(mode: DomainRepeatMode) = repository.setRepeatMode(mode)
    suspend fun setShuffleMode(mode: DomainShuffleMode) = repository.setShuffleMode(mode)
    suspend fun setScrubbing(isScrubbing: Boolean) = repository.setScrubbing(isScrubbing)

    fun getAudioSessionId(): Int? = repository.getPlayerSessionId()

    // Helper to load a fresh playlist (replaces existing queue)
    suspend fun loadAndPlay(
        items: List<PlaybackItem>,
        startIndex: Int = 0,
        startPositionMs: Long = 0L,
        shouldShuffle: Boolean = false
    ) {
        if (items.isEmpty()) return
        repository.prepareAndPlay(items, startIndex, startPositionMs, shouldShuffle)
    }
}

// File: java\com\example\holodex\playback\domain\usecase\PlayItemsUseCase.kt
// File: java/com/example/holodex/playback/domain/usecase/PlayItemsUseCase.kt
package com.example.holodex.playback.domain.usecase

import com.example.holodex.playback.domain.model.PlaybackItem
import com.example.holodex.playback.domain.repository.PlaybackRepository

class PlayItemsUseCase(private val playbackRepository: PlaybackRepository) {
    suspend operator fun invoke(
        items: List<PlaybackItem>,
        startIndex: Int = 0,
        startPositionMs: Long = 0L,
        shouldShuffle: Boolean = false
    ) {
        if (items.isEmpty()) return
```

```
        playbackRepository.prepareAndPlay(items, startIndex, startPositionMs, shouldShuffle)
    }
}

// File: java\com\example\holodex\playback\domain\usecase\QueueManagementUseCase.kt
package com.example.holodex.playback.domain.usecase

import com.example.holodex.playback.domain.model.PlaybackItem
import com.example.holodex.playback.domain.repository.PlaybackRepository
import javax.inject.Inject

class QueueManagementUseCase @Inject constructor(
    private val repository: PlaybackRepository
) {
    suspend fun add(item: PlaybackItem) = repository.addItemToQueue(item, null)
    suspend fun addAll(items: List<PlaybackItem>) = repository.addItemsToQueue(items, null)
    suspend fun remove(index: Int) = repository.removeItemFromQueue(index)
    suspend fun move(from: Int, to: Int) = repository.reorderQueueItem(from, to)
    suspend fun clear() = repository.clearQueue()
}

// File: java\com\example\holodex\playback\domain\usecase\ReleasePlaybackResourcesUseCase.kt
package com.example.holodex.playback.domain.usecase

import com.example.holodex.playback.domain.repository.PlaybackRepository


class ReleasePlaybackResourcesUseCase(private val playbackRepository: PlaybackRepository) {
    operator fun invoke() { // Not suspend as release might be a synchronous cleanup
        playbackRepository.release()
    }
}

// File: java\com\example\holodex\playback\domain\usecase\RemoveItemFromQueueUseCase.kt
package com.example.holodex.playback.domain.usecase

import com.example.holodex.playback.domain.repository.PlaybackRepository

class RemoveItemFromQueueUseCase(private val playbackRepository: PlaybackRepository) {
    suspend operator fun invoke(index: Int) {
        playbackRepository.removeItemFromQueue(index)
    }
}

// File: java\com\example\holodex\playback\domain\usecase\ReorderQueueItemUseCase.kt
package com.example.holodex.playback.domain.usecase

import com.example.holodex.playback.domain.repository.PlaybackRepository


class ReorderQueueItemUseCase(private val playbackRepository: PlaybackRepository) {
    suspend operator fun invoke(fromIndex: Int, toIndex: Int) {
        playbackRepository.reorderQueueItem(fromIndex, toIndex)
    }
}
```

```
// File: java\com\example\holodex\playback\domain\usecase\ResolveStreamUrlUseCase.kt
package com.example.holodex.playback.domain.usecase

import com.example.holodex.playback.domain.model.StreamDetails
import com.example.holodex.playback.domain.repository.StreamResolverRepository


class ResolveStreamUrlUseCase(
    private val streamResolverRepository: StreamResolverRepository
) {
    suspend operator fun invoke(videoId: String): Result<StreamDetails> {
        return streamResolverRepository.resolveStreamUrl(videoId)
    }
}

// File: java\com\example\holodex\playback\domain\usecase\ResumePlaybackUseCase.kt
package com.example.holodex.playback.domain.usecase

import com.example.holodex.playback.domain.repository.PlaybackRepository

class ResumePlaybackUseCase(private val playbackRepository: PlaybackRepository) {
    suspend operator fun invoke() {
        // 'play' can often serve as resume if the player is already prepared
        playbackRepository.play()
    }
}

// File: java\com\example\holodex\playback\domain\usecase\SavePlaybackStateUseCase.kt
package com.example.holodex.playback.domain.usecase

import com.example.holodex.playback.domain.model.PersistedPlaybackData
import com.example.holodex.playback.domain.repository.PlaybackStateRepository

class SavePlaybackStateUseCase(
    private val playbackStateRepository: PlaybackStateRepository
) {
    suspend operator fun invoke(data: PersistedPlaybackData) {
        playbackStateRepository.saveState(data)
    }
}

// File: java\com\example\holodex\playback\domain\usecase\SeekPlaybackUseCase.kt
// File: holodex\playback\domain\usecase\SeekPlaybackUseCase.kt
package com.example.holodex.playback.domain.usecase

import com.example.holodex.playback.domain.repository.PlaybackRepository

class SeekPlaybackUseCase(private val playbackRepository: PlaybackRepository) {
    suspend operator fun invoke(positionSec: Long) { // Changed parameter to Sec
        playbackRepository.seekTo(positionSec)
    }
}

// File: java\com\example\holodex\playback\domain\usecase\SetRepeatModeUseCase.kt
```

```
package com.example.holodex.playback.domain.usecase

import com.example.holodex.playback.domain.model.DomainRepeatMode
import com.example.holodex.playback.domain.repository.PlaybackRepository


class SetRepeatModeUseCase(private val playbackRepository: PlaybackRepository) {
    suspend operator fun invoke(mode: DomainRepeatMode) {
        playbackRepository.setRepeatMode(mode)
    }
}

// File: java\com\example\holodex\playback\domain\usecase\SetScrubbingUseCase.kt
package com.example.holodex.playback.domain.usecase

import com.example.holodex.playback.domain.repository.PlaybackRepository

class SetScrubbingUseCase(private val playbackRepository: PlaybackRepository) {
    suspend operator fun invoke(isScrubbing: Boolean) {
        playbackRepository.setScrubbing(isScrubbing)
    }
}

// File: java\com\example\holodex\playback\domain\usecase\SetShuffleModeUseCase.kt
package com.example.holodex.playback.domain.usecase

import com.example.holodex.playback.domain.model.DomainShuffleMode
import com.example.holodex.playback.domain.repository.PlaybackRepository


class SetShuffleModeUseCase(private val playbackRepository: PlaybackRepository) {
    suspend operator fun invoke(mode: DomainShuffleMode) {
        playbackRepository.setShuffleMode(mode)
    }
}

// File: java\com\example\holodex\playback\domain\usecase\SkipToNextItemUseCase.kt
package com.example.holodex.playback.domain.usecase

import com.example.holodex.playback.domain.repository.PlaybackRepository


class SkipToNextItemUseCase(private val playbackRepository: PlaybackRepository) {
    suspend operator fun invoke() {
        playbackRepository.skipToNext()
    }
}

// File: java\com\example\holodex\playback\domain\usecase\SkipToPreviousItemUseCase.kt
package com.example.holodex.playback.domain.usecase

import com.example.holodex.playback.domain.repository.PlaybackRepository

class SkipToPreviousItemUseCase(private val playbackRepository: PlaybackRepository) {
    suspend operator fun invoke() {
```

```
        playbackRepository.skipToPrevious()
    }
}

// File: java\com\example\holodex\playback\domain\usecase\SkipToQueueItemUseCase.kt
// File: java/com/example/holodex/playback/domain/usecase/SkipToQueueItemUseCase.kt

package com.example.holodex.playback.domain.usecase

import com.example.holodex.playback.domain.repository.PlaybackRepository

class SkipToQueueItemUseCase(private val playbackRepository: PlaybackRepository) {
    suspend operator fun invoke(index: Int) {
        playbackRepository.skipToQueueItem(index)
    }
}

// File: java\com\example\holodex\playback\player\Media3PlayerController.kt
// File: java/com/example/holodex/playback/player/Media3PlayerController.kt
package com.example.holodex.playback.player

import androidx.media3.common.AudioAttributes
import androidx.media3.common.C
import androidx.media3.common.MediaItem
import androidx.media3.common.PlaybackException
import androidx.media3.common.Player
import androidx.media3.common.Timeline
import androidx.media3.common.util.UnstableApi
import androidx.media3.exoplayer.ExoPlayer
import androidx.media3.exoplayer.source.preload.DefaultPreloadManager
import androidx.media3.exoplayer.util.EventLogger
import com.example.holodex.playback.domain.model.DomainPlaybackState
import com.example.holodex.playback.util.PlayerStateMapper
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.SupervisorJob
import kotlinx.coroutines.flow.MutableSharedFlow
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.SharedFlow
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asSharedFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.launch
import timber.log.Timber

data class PlayerMediaItemTransition(val mediaItem: MediaItem?, val newIndex: Int, val reason:
data class PlayerErrorEvent(val error: PlaybackException)
data class PlayerTimelineChangedEvent(val timeline: Timeline, val reason: Int)
data class PlayerIsPlayingChangedEvent(val isPlaying: Boolean)
data class PlayerDiscontinuityEvent(
    val oldPosition: Player.PositionInfo,
    val newPosition: Player.PositionInfo,
    val reason: Int
)
```

```
@UnstableApi
class Media3PlayerController(
    val exoPlayer: ExoPlayer,
    private val preloadManager: DefaultPreloadManager
) : Player.Listener {

    companion object {
        private const val TAG = "Media3PlayerController"
    }

    private val eventLogger: EventLogger = EventLogger(TAG)
    private val controllerScope = CoroutineScope(Dispatchers.Main.immediate + SupervisorJob())

    private val _playerPlaybackStateFlow = MutableStateFlow(DomainPlaybackState.IDLE)
    val playerPlaybackStateFlow: StateFlow<DomainPlaybackState> = _playerPlaybackStateFlow.asS

    private val _mediaItemTransitionEventFlow = MutableSharedFlow<PlayerMediaItemTransition>(r
    val mediaItemTransitionEventFlow: SharedFlow<PlayerMediaItemTransition> = _mediaItemTransi

    private val _isPlayingChangedEventFlow = MutableSharedFlow<PlayerIsPlayingChangedEvent>(re
    val isPlayingChangedEventFlow: SharedFlow<PlayerIsPlayingChangedEvent> = _isPlayingChanged

    private val _discontinuityEventFlow = MutableSharedFlow<PlayerDiscontinuityEvent>(replay =
    val discontinuityEventFlow: SharedFlow<PlayerDiscontinuityEvent> = _discontinuityEventFlow

    init {
        exoPlayer.addListener(this)
        exoPlayer.addAnalyticsListener(eventLogger)
        setupAudioAttributes()

        _playerPlaybackStateFlow.value = PlayerStateMapper.mapExoPlayerStateToDomain(exoPlayer
    }

    private fun setupAudioAttributes() {
        val audioAttributes = AudioAttributes.Builder()
            .setContentType(C.AUDIO_CONTENT_TYPE_MUSIC)
            .setUsage(C.USAGE_MEDIA)
            .build()
        exoPlayer.setAudioAttributes(audioAttributes, true)
    }

    fun play() {
        if (exoPlayer.playbackState == Player.STATE_IDLE && exoPlayer.mediaItemCount > 0) {
            exoPlayer.prepare()
        }
        if (exoPlayer.playbackState == Player.STATE_ENDED) {
            exoPlayer.seekToDefaultPosition()
        }
        exoPlayer.play()
    }

    fun stop() {
        exoPlayer.stop()
        exoPlayer.clearMediaItems()
    }
```

```
fun pause() = exoPlayer.pause()
fun seekTo(positionMs: Long) {
    if (exoPlayer.isCommandAvailable(Player.COMMAND_SEEK_IN_CURRENT_MEDIA_ITEM)) {
        exoPlayer.seekTo(positionMs.coerceAtLeast(0L))
    }
}

fun seekToItem(itemIndex: Int, positionMs: Long) {
    if (exoPlayer.isCommandAvailable(Player.COMMAND_SEEK_TO_MEDIA_ITEM)) {
        exoPlayer.seekTo(itemIndex, positionMs.coerceAtLeast(0L))
    }
}

fun setRepeatMode(@Player.RepeatMode exoPlayerMode: Int) {
    exoPlayer.repeatMode = exoPlayerMode
}

fun setMediaItems(items: List<MediaItem>, startIndex: Int, startPositionMs: Long) {
    updatePreloadManagerPlaylist(items)
    if (items.isNotEmpty()) {
        exoPlayer.setMediaItems(items, startIndex, startPositionMs)
        exoPlayer.prepare()
    } else {
        _playerPlaybackStateFlow.value = PlayerStateMapper.mapExoPlayerStateToDomain(exoPl
    }
}

fun clearMediaItemsAndStop() {
    exoPlayer.stop()
    exoPlayer.clearMediaItems()
    preloadManager.reset()
}

internal  fun getMediaItemsFromPlayerTimeline(): List<MediaItem> {
    val timeline = exoPlayer.currentTimeline
    if (timeline.isEmpty) return emptyList()
    return (0 until timeline.windowCount).map {
        timeline.getWindow(it, Timeline.Window()).mediaItem
    }
}

private fun updatePreloadManagerPlaylist(mediaItems: List<MediaItem>) {
    try {
        preloadManager.reset()
        if (mediaItems.isEmpty()) {
            preloadManager.setCurrentPlayingIndex(C.INDEX_UNSET)
            return
        }
        val preloadableItems = mediaItems.filter { it.localConfiguration?.uri?.scheme != "
        preloadableItems.forEach { item ->
            val originalIndex = mediaItems.indexOf(item)
            if (originalIndex != -1) {
                preloadManager.add(item, originalIndex)
            }
```

```
            }
            preloadManager.setCurrentPlayingIndex(exoPlayer.currentMediaItemIndex)
        } catch (e: Exception) {
            Timber.e(e, "$TAG: Error updating preload manager playlist.")
        }
    }

    fun releasePlayer() {
        exoPlayer.removeAnalyticsListener(eventLogger)
        exoPlayer.removeListener(this)
        exoPlayer.release()
    }

    override fun onPlaybackStateChanged(playbackState: Int) = onStateChanged()
    override fun onPlayWhenReadyChanged(playWhenReady: Boolean, reason: Int) = onStateChanged(
    private fun onStateChanged() {
        _playerPlaybackStateFlow.value = PlayerStateMapper.mapExoPlayerStateToDomain(exoPlayer
    }

    override fun onIsPlayingChanged(isPlaying: Boolean) {
        controllerScope.launch { _isPlayingChangedEventFlow.emit(PlayerIsPlayingChangedEvent(i
    }

    override fun onMediaItemTransition(mediaItem: MediaItem?, reason: Int) {
        val newIndex = exoPlayer.currentMediaItemIndex.takeIf { it != C.INDEX_UNSET } ?: -1
        controllerScope.launch { _mediaItemTransitionEventFlow.emit(PlayerMediaItemTransition(
    }

    override fun onTimelineChanged(timeline: Timeline, reason: Int) {
        if (reason == Player.TIMELINE_CHANGE_REASON_PLAYLIST_CHANGED) {
            updatePreloadManagerPlaylist(getMediaItemsFromPlayerTimeline())
        }
    }

    override fun onPlayerError(error: PlaybackException) {
        _playerPlaybackStateFlow.value = DomainPlaybackState.ERROR
    }

    override fun onPositionDiscontinuity(oldPosition: Player.PositionInfo, newPosition: Player
        controllerScope.launch { _discontinuityEventFlow.emit(PlayerDiscontinuityEvent(oldPosi
    }

    fun updatePreloadIndex(newIndex: Int) {
        try {
            preloadManager.setCurrentPlayingIndex(newIndex)
        } catch (e: Exception) {
            Timber.e(e, "$TAG: Error updating preload manager index")
        }
    }
}

// File: java\com\example\holodex\playback\player\MediaControllerManager.kt
// File: java/com/example/holodex/playback/player/MediaControllerManager.kt (NEW FILE)
package com.example.holodex.playback.player
```

```
import android.content.ComponentName
import android.content.Context
import androidx.media3.session.MediaController
import androidx.media3.session.SessionToken
import com.example.holodex.service.MediaPlaybackService
import com.google.common.util.concurrent.ListenableFuture
import com.google.common.util.concurrent.MoreExecutors
import dagger.hilt.android.qualifiers.ApplicationContext
import kotlinx.coroutines.CompletableDeferred
import timber.log.Timber
import javax.inject.Inject
import javax.inject.Singleton

/**
 * A Singleton manager that provides a single, app-wide instance of MediaController.
 * It handles the asynchronous connection to the MediaSessionService.
 */
@Singleton
class MediaControllerManager @Inject constructor(
    @ApplicationContext private val context: Context
) {
    private val sessionToken = SessionToken(context, ComponentName(context, MediaPlaybackServi
    private val controllerFuture: ListenableFuture<MediaController> =
        MediaController.Builder(context, sessionToken).buildAsync()

    private val deferredController = CompletableDeferred<MediaController>()

    init {
        controllerFuture.addListener({
            try {
                val controller = controllerFuture.get()
                deferredController.complete(controller)
                Timber.d("MediaControllerManager: MediaController connected successfully.")
            } catch (e: Exception) {
                deferredController.completeExceptionally(e)
                Timber.e(e, "MediaControllerManager: Failed to connect MediaController.")
            }
        }, MoreExecutors.directExecutor())
    }

    /**
     * Suspends until the MediaController is connected, then returns the instance.
     */
    suspend fun awaitController(): MediaController {
        return deferredController.await()
    }

    fun release() {
        if (controllerFuture.isDone) {
            MediaController.releaseFuture(controllerFuture)
        }
    }
}

// File: java\com\example\holodex\playback\player\TimelineSynchronizer.kt
```

```
package com.example.holodex.playback.player

import androidx.media3.exoplayer.ExoPlayer
import com.example.holodex.playback.data.queue.PlaybackQueueState
import com.example.holodex.playback.domain.model.DomainShuffleMode
import timber.log.Timber
import javax.inject.Inject

/**
 * Responsible for keeping the ExoPlayer timeline in sync with the PlaybackQueueState.
 * It calculates the minimal set of moves required to make the player match the domain state.
 */
class TimelineSynchronizer @Inject constructor() {

    private data class Move(val from: Int, val to: Int)

    suspend fun syncPlayerWithQueueState(player: ExoPlayer, queueState: PlaybackQueueState) {
        // 1. Sync Shuffle Mode
        val shouldShuffle = queueState.shuffleMode == DomainShuffleMode.ON
        if (player.shuffleModeEnabled != shouldShuffle) {
            player.shuffleModeEnabled = shouldShuffle
        }

        // 2. Sync Timeline Order
        // We map IDs to find discrepancies between the Player's reality and the Queue's reali
        val currentTimelineIds = getMediaIdsFromPlayer(player)
        val desiredOrderIds = queueState.activeList.map { it.id }

        if (currentTimelineIds == desiredOrderIds) {
            return // Perfect match, no work needed
        }

        // Calculate and apply moves
        val moves = calculateOptimalMoves(currentTimelineIds, desiredOrderIds)

        // Apply moves sequentially
        moves.forEach { move ->
            if (move.from in 0 until player.mediaItemCount && move.to in 0 until player.mediaI
                player.moveMediaItem(move.from, move.to)
            } else {
                Timber.e("TimelineSynchronizer: Invalid move calculated: $move. Player size: $
            }
        }
    }

    private fun getMediaIdsFromPlayer(player: ExoPlayer): List<String> {
        val timeline = player.currentTimeline
        if (timeline.isEmpty) return emptyList()
        return (0 until timeline.windowCount).map {
            // We assume mediaId is set correctly on all items
            player.getMediaItemAt(it).mediaId
        }
    }

    /**
```

```
     * Calculates the moves needed to transform [current] list into [desired] list.
     */
    private fun calculateOptimalMoves(current: List<String>, desired: List<String>): List<Move
        if (current.size != desired.size) {
            // If sizes mismatch, we can't just reorder. The Repository should handle adding/r
            // This checks prevents crashes.
            return emptyList()
        }

        val moves = mutableListOf<Move>()
        val workingList = current.toMutableList()

        for (targetIndex in desired.indices) {
            val targetId = desired[targetIndex]
            val currentIndex = workingList.indexOf(targetId)

            if (currentIndex != -1 && currentIndex != targetIndex) {
                moves.add(Move(from = currentIndex, to = targetIndex))
                val item = workingList.removeAt(currentIndex)
                workingList.add(targetIndex, item)
            }
        }
        return moves
    }
}

// File: java\com\example\holodex\playback\util\PlaybackUtil.kt
// File: java/com/example/holodex/playback/util/PlaybackUtil.kt
package com.example.holodex.playback.util

import androidx.media3.common.Player
import java.util.Locale
import java.util.concurrent.TimeUnit

/** Returns a human-readable name for the given playback state. */
fun playbackStateToString(state: Int): String = when (state) {
    Player.STATE_IDLE -> "STATE_IDLE"
    Player.STATE_BUFFERING -> "STATE_BUFFERING"
    Player.STATE_READY -> "STATE_READY"
    Player.STATE_ENDED -> "STATE_ENDED"
    else -> "UNKNOWN_PLAYBACK_STATE($state)"
}

/** Returns a readable name for media-item transition reasons. */
fun mediaItemTransitionReasonToString(reason: Int): String = when (reason) {
    Player.MEDIA_ITEM_TRANSITION_REASON_AUTO -> "AUTO"
    Player.MEDIA_ITEM_TRANSITION_REASON_PLAYLIST_CHANGED -> "PLAYLIST_CHANGED"
    Player.MEDIA_ITEM_TRANSITION_REASON_REPEAT -> "REPEAT_MODE"
    Player.MEDIA_ITEM_TRANSITION_REASON_SEEK -> "SEEK"
    else -> "UNKNOWN_TRANSITION_REASON($reason)"
}

/** Returns a readable name for timeline-change reasons. */
fun timelineChangeReasonToString(reason: Int): String = when (reason) {
    Player.TIMELINE_CHANGE_REASON_PLAYLIST_CHANGED -> "PLAYLIST_CHANGED"
```

```
    Player.TIMELINE_CHANGE_REASON_SOURCE_UPDATE -> "SOURCE_UPDATE"
    else -> "UNKNOWN_TIMELINE_REASON($reason)"
}

/** Returns a readable name for discontinuity reasons. */
fun discontinuityReasonToString(reason: Int): String = when (reason) {
    Player.DISCONTINUITY_REASON_AUTO_TRANSITION -> "AUTO_TRANSITION"
    Player.DISCONTINUITY_REASON_SEEK -> "SEEK"
    Player.DISCONTINUITY_REASON_SEEK_ADJUSTMENT -> "SEEK_ADJUSTMENT"
    Player.DISCONTINUITY_REASON_REMOVE -> "REMOVE"
    Player.DISCONTINUITY_REASON_INTERNAL -> "INTERNAL"
    else -> "UNKNOWN_DISCONTINUITY_REASON($reason)"
}

/** Formats a duration in seconds into a MM:SS string. */
fun formatSongTimestamp(seconds: Long): String {
    if (seconds < 0) return "--:--"
    val minutes = TimeUnit.SECONDS.toMinutes(seconds) % 60
    val secs = seconds % 60
    return String.format(Locale.US, "%02d:%02d", minutes, secs)
}

/** Formats a total duration in seconds into a HH:MM:SS or MM:SS string. */
fun formatDurationSecondsToString(totalSeconds: Long): String {
    if (totalSeconds < 0) return "--:--"
    if (totalSeconds == 0L) return "00:00"

    val hours = TimeUnit.SECONDS.toHours(totalSeconds)
    val minutes = TimeUnit.SECONDS.toMinutes(totalSeconds) % 60
    val secs = totalSeconds % 60

    return if (hours > 0) {
        String.format(Locale.US, "%d:%02d:%02d", hours, minutes, secs)
    } else {
        String.format(Locale.US, "%02d:%02d", minutes, secs)
    }
}

/** Formats a total duration in seconds into a HH:MM:SS or MM:SS string. */
fun formatDurationSeconds(totalSecondsLong: Long): String {
    if (totalSecondsLong <= 0) return ""
    val hours = TimeUnit.SECONDS.toHours(totalSecondsLong)
    val minutes = TimeUnit.SECONDS.toMinutes(totalSecondsLong) % 60
    val secs = totalSecondsLong % 60

    return if (hours > 0) {
        String.format(Locale.US, "%d:%02d:%02d", hours, minutes, secs)
    } else {
        String.format(Locale.US, "%02d:%02d", minutes, secs)
    }
}

// File: java\com\example\holodex\playback\util\PlayerStateMapper.kt
// File: java/com/example/holodex/playback/util/PlayerStateMapper.kt
package com.example.holodex.playback.util
```

```
import androidx.media3.common.Player
import com.example.holodex.playback.domain.model.DomainPlaybackState
import com.example.holodex.playback.domain.model.DomainRepeatMode
import com.example.holodex.playback.domain.model.DomainShuffleMode

object PlayerStateMapper {

    fun mapExoPlayerStateToDomain(playbackState: Int, playWhenReady: Boolean): DomainPlaybackS
        return when (playbackState) {
            Player.STATE_IDLE -> DomainPlaybackState.IDLE
            Player.STATE_BUFFERING -> DomainPlaybackState.BUFFERING
            Player.STATE_READY -> if (playWhenReady) DomainPlaybackState.PLAYING else DomainPl
            Player.STATE_ENDED -> DomainPlaybackState.ENDED
            else -> DomainPlaybackState.IDLE
        }
    }
    @Player.RepeatMode
    fun mapDomainRepeatModeToExoPlayer(domainMode: DomainRepeatMode): Int {
        return when (domainMode) {
            DomainRepeatMode.NONE -> Player.REPEAT_MODE_OFF
            DomainRepeatMode.ONE -> Player.REPEAT_MODE_ONE
            DomainRepeatMode.ALL -> Player.REPEAT_MODE_ALL
        }
    }

    fun mapExoPlayerRepeatModeToDomain(@Player.RepeatMode exoPlayerMode: Int): DomainRepeatMod
        return when (exoPlayerMode) {
            Player.REPEAT_MODE_OFF -> DomainRepeatMode.NONE
            Player.REPEAT_MODE_ONE -> DomainRepeatMode.ONE
            Player.REPEAT_MODE_ALL -> DomainRepeatMode.ALL
            else -> DomainRepeatMode.NONE
        }
    }

    fun mapDomainShuffleModeToExoPlayer(domainMode: DomainShuffleMode): Boolean {
        return domainMode == DomainShuffleMode.ON
    }

    fun mapExoPlayerShuffleModeToDomain(exoPlayerShuffleEnabled: Boolean): DomainShuffleMode {
        return if (exoPlayerShuffleEnabled) DomainShuffleMode.ON else DomainShuffleMode.OFF
    }
}

// File: java\com\example\holodex\service\HolodexDownloadService.kt
package com.example.holodex.service

import android.app.Notification
import androidx.annotation.OptIn
import androidx.media3.common.util.UnstableApi
import androidx.media3.exoplayer.offline.Download
import androidx.media3.exoplayer.offline.DownloadManager
import androidx.media3.exoplayer.offline.DownloadNotificationHelper
import androidx.media3.exoplayer.offline.DownloadService
import androidx.media3.exoplayer.scheduler.Scheduler
```

```
import androidx.media3.exoplayer.workmanager.WorkManagerScheduler
import com.example.holodex.R
import dagger.hilt.android.AndroidEntryPoint
import timber.log.Timber
import javax.inject.Inject

@AndroidEntryPoint
@OptIn(UnstableApi::class)
class HolodexDownloadService : DownloadService(
    FOREGROUND_NOTIFICATION_ID,
    DEFAULT_FOREGROUND_NOTIFICATION_UPDATE_INTERVAL,
    DOWNLOAD_NOTIFICATION_CHANNEL_ID,
    R.string.download_notification_channel_name,
    R.string.download_notification_channel_description
) {

    // --- Injected fields remain the same ---
    @Inject
    lateinit var downloadManagerInstance: DownloadManager

    @Inject
    lateinit var notificationHelper: DownloadNotificationHelper

    // --- All listener and scope code is REMOVED from here down to onDestroy ---

    companion object {
        private const val FOREGROUND_NOTIFICATION_ID = 2
        private const val DOWNLOAD_NOTIFICATION_CHANNEL_ID = "download_channel"
        private const val DOWNLOAD_WORK_MANAGER_JOB_ID = "holodex_download_job"
    }

    override fun getDownloadManager(): DownloadManager {
        // Just return the injected instance. No need to add/remove listeners.
        return downloadManagerInstance
    }

    override fun getScheduler(): Scheduler {
        return WorkManagerScheduler(this, DOWNLOAD_WORK_MANAGER_JOB_ID)
    }

    override fun getForegroundNotification(
        downloads: MutableList<Download>,
        notMetRequirements: Int
    ): Notification {
        return notificationHelper.buildProgressNotification(
            this, R.drawable.ic_notification_small, null, null, downloads, notMetRequirements
        )
    }

    // --- No need for onDestroy to cancel a job anymore ---
    override fun onDestroy() {
        super.onDestroy()
        Timber.d("HolodexDownloadService destroyed.")
    }
}
```

```
// File: java\com\example\holodex\service\MediaPlaybackService.kt
// File: java/com/example/holodex/service/MediaPlaybackService.kt
package com.example.holodex.service

import android.app.Notification
import android.app.PendingIntent
import android.content.Intent
import android.graphics.Bitmap
import android.graphics.Canvas
import android.graphics.drawable.BitmapDrawable
import android.graphics.drawable.Drawable
import android.os.Build
import android.os.Bundle
import androidx.core.content.ContextCompat
import androidx.core.graphics.createBitmap
import androidx.media3.common.Player
import androidx.media3.common.util.UnstableApi
import androidx.media3.session.MediaSession
import androidx.media3.session.MediaSession.ControllerInfo
import androidx.media3.session.MediaSessionService
import androidx.media3.session.SessionCommand
import androidx.media3.session.SessionCommands
import androidx.media3.session.SessionError
import androidx.media3.session.SessionResult
import androidx.media3.ui.PlayerNotificationManager
import coil.imageLoader
import coil.request.ImageRequest
import com.example.holodex.R
import com.example.holodex.playback.domain.model.PlaybackItem
import com.example.holodex.playback.domain.usecase.PlayItemsUseCase
import com.example.holodex.playback.util.playbackStateToString
import com.example.holodex.ui.MainActivity
import com.google.common.util.concurrent.Futures
import com.google.common.util.concurrent.ListenableFuture
import dagger.hilt.android.AndroidEntryPoint
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.SupervisorJob
import kotlinx.coroutines.cancel
import kotlinx.coroutines.guava.future
import timber.log.Timber
import javax.inject.Inject

// --- Utility function for Bitmap conversion ---
fun Drawable?.toBitmapSafe(): Bitmap {
    if (this == null) {
        Timber.tag("BmpSafe").w("Drawable was null, returning 1x1 bitmap.")
        return createBitmap(1, 1, Bitmap.Config.ARGB_8888)
    }
    if (this is BitmapDrawable && this.bitmap != null) { return this.bitmap }
    val width = intrinsicWidth.coerceAtLeast(1)
    val height = intrinsicHeight.coerceAtLeast(1)
    val bitmap = createBitmap(width, height, Bitmap.Config.ARGB_8888)
    val canvas = Canvas(bitmap)
```

```
    this.setBounds(0, 0, canvas.width, canvas.height)
    this.draw(canvas)
    return bitmap
}

// --- Constants ---
const val CUSTOM_COMMAND_PREPARE_FROM_REQUEST = "com.example.holodex.PREPARE_FROM_REQUEST"
const val ARG_PLAYBACK_ITEMS_LIST = "playback_items_list"
const val ARG_START_INDEX = "start_index"
const val ARG_START_POSITION_SEC = "start_position_sec"
const val ARG_SHOULD_SHUFFLE = "should_shuffle_playlist"

private const val SERVICE_NOTIFICATION_ID = 123
private const val PLAYBACK_NOTIFICATION_CHANNEL_ID = "holodex_playback_channel_v3"
private const val SERVICE_TAG = "MediaPlaybackService"

@UnstableApi
@AndroidEntryPoint
class MediaPlaybackService : MediaSessionService() {

    private var mediaSession: MediaSession? = null
    private val serviceJob = SupervisorJob()
    private val serviceScope = CoroutineScope(Dispatchers.Main.immediate + serviceJob)

    @Inject lateinit var player: Player
    @Inject lateinit var playItemsUseCase: PlayItemsUseCase

    private lateinit var notificationManager: PlayerNotificationManager
    private var defaultNotificationBitmap: Bitmap? = null
    private var isServiceInForeground = false

    private val playerListener = PlayerStateListener()

    override fun onCreate() {
        super.onCreate()
        Timber.tag(SERVICE_TAG).i("onCreate: Service creating...")

        defaultNotificationBitmap = (ContextCompat.getDrawable(this, R.drawable.ic_default_alb
            ?.toBitmapSafe()) ?: createBitmap(64, 64, Bitmap.Config.ARGB_8888)

        player.addListener(playerListener)

        initializeMediaSession()
        initializeNotificationManager()
        Timber.tag(SERVICE_TAG).i("onCreate: Service creation complete.")
    }

    private fun initializeMediaSession() {
        mediaSession = MediaSession.Builder(this, player)
            .setSessionActivity(getSingleTopActivityPendingIntent())
            .setCallback(MediaSessionCallback())
            .setId("holodex_music_media_session_${System.currentTimeMillis()}")
            .build()

        addSession(mediaSession!!)
```

```
}

private fun initializeNotificationManager() {
    val mediaDescriptionAdapter = ServiceMediaDescriptionAdapter()
    val notificationListener = ServiceNotificationListener()

    notificationManager = PlayerNotificationManager.Builder(
        this,
        SERVICE_NOTIFICATION_ID,
        PLAYBACK_NOTIFICATION_CHANNEL_ID
    )
        .setChannelNameResourceId(R.string.playback_notification_channel_name)
        .setChannelDescriptionResourceId(R.string.playback_notification_channel_descriptio
        .setMediaDescriptionAdapter(mediaDescriptionAdapter)
        .setNotificationListener(notificationListener)
        .setSmallIconResourceId(R.drawable.ic_stat_music_note)
        .build().apply {
            setUseRewindAction(false)
            setUseFastForwardAction(false)
            setUseNextAction(true)
            setUsePreviousAction(true)
            setColorized(true)
            setUseNextActionInCompactView(true)
            setUsePreviousActionInCompactView(true)
            setPlayer(player)
        }
}

private fun getSingleTopActivityPendingIntent(): PendingIntent {
    val intent = Intent(this, MainActivity::class.java).apply {
        flags = Intent.FLAG_ACTIVITY_SINGLE_TOP or Intent.FLAG_ACTIVITY_CLEAR_TOP
    }
    val flags = PendingIntent.FLAG_IMMUTABLE or PendingIntent.FLAG_UPDATE_CURRENT
    return PendingIntent.getActivity(this, 0, intent, flags)
}

@Suppress("OVERRIDE_DEPRECATION")
override fun onUpdateNotification(session: MediaSession, startInForegroundRequired: Boolea
    // Deprecated but still called. Handled by PlayerNotificationManager.
}

private inner class PlayerStateListener : Player.Listener {
    override fun onPlaybackStateChanged(playbackState: Int) {
        Timber.tag(SERVICE_TAG).i("PlayerListener.onPlaybackStateChanged: %s", playbackSta
    }

    override fun onIsPlayingChanged(isPlaying: Boolean) {
        Timber.tag(SERVICE_TAG).i("PlayerListener.onIsPlayingChanged: %b", isPlaying)
    }
}

private inner class ServiceMediaDescriptionAdapter : PlayerNotificationManager.MediaDescri
    override fun getCurrentContentTitle(player: Player): CharSequence {
        return player.currentMediaItem?.mediaMetadata?.title ?: getString(R.string.unknown
    }
```

```
    override fun createCurrentContentIntent(player: Player): PendingIntent? {
        return getSingleTopActivityPendingIntent()
    }

    override fun getCurrentContentText(player: Player): CharSequence? {
        return player.currentMediaItem?.mediaMetadata?.artist
    }

    override fun getCurrentLargeIcon(player: Player, callback: PlayerNotificationManager.B
        val artworkUri = player.currentMediaItem?.mediaMetadata?.artworkUri
        if (artworkUri != null) {
            val request = ImageRequest.Builder(applicationContext)
                .data(artworkUri)
                .allowHardware(false)
                .target(
                    onSuccess = { drawable -> callback.onBitmap(drawable.toBitmapSafe()) }
                    onError = { defaultNotificationBitmap?.let { callback.onBitmap(it) } }
                ).build()
            applicationContext.imageLoader.enqueue(request)
            return defaultNotificationBitmap
        }
        return defaultNotificationBitmap
    }
}

private inner class ServiceNotificationListener : PlayerNotificationManager.NotificationLi
    override fun onNotificationPosted(notificationId: Int, notification: Notification, ong
        if (ongoing) {
            if (!isServiceInForeground) {
                try {
                    startForeground(notificationId, notification)
                    isServiceInForeground = true
                } catch (e: Exception) {
                    Timber.e(e, "CRITICAL EXCEPTION during startForeground().")
                }
            }
        } else {
            if (isServiceInForeground) {
                stopForeground(STOP_FOREGROUND_REMOVE)
                isServiceInForeground = false
            }
        }
    }

    override fun onNotificationCancelled(notificationId: Int, dismissedByUser: Boolean) {
        if (dismissedByUser) {
            player.stop()
            stopSelf()
        }
        isServiceInForeground = false
    }
}

override fun onGetSession(controllerInfo: ControllerInfo): MediaSession = mediaSession!!
```

```
private inner class MediaSessionCallback : MediaSession.Callback {
    override fun onConnect(session: MediaSession, controller: ControllerInfo): MediaSessio
        val sessionCommands = SessionCommands.Builder()
            .add(SessionCommand(CUSTOM_COMMAND_PREPARE_FROM_REQUEST, Bundle.EMPTY))
            .build()
        return MediaSession.ConnectionResult.accept(sessionCommands, player.availableComma
    }

    override fun onCustomCommand(
        session: MediaSession,
        controller: ControllerInfo,
        customCommand: SessionCommand,
        args: Bundle
    ): ListenableFuture<SessionResult> {
        if (customCommand.customAction == CUSTOM_COMMAND_PREPARE_FROM_REQUEST) {
            val itemsParcelable: ArrayList<PlaybackItem>? =
                if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
                    args.getParcelableArrayList(ARG_PLAYBACK_ITEMS_LIST, PlaybackItem::cla
                } else {
                    @Suppress("DEPRECATION")
                    args.getParcelableArrayList(ARG_PLAYBACK_ITEMS_LIST)
                }
            val startIndex = args.getInt(ARG_START_INDEX, 0)
            val startPositionSec = args.getLong(ARG_START_POSITION_SEC, 0L)
            val shouldShufflePlaylist = args.getBoolean(ARG_SHOULD_SHUFFLE, false)

            if (itemsParcelable.isNullOrEmpty()) {
                return Futures.immediateFuture(SessionResult(SessionError.ERROR_BAD_VALUE)
            }

            return serviceScope.future {
                try {
                    playItemsUseCase(itemsParcelable, startIndex, startPositionSec * 1000L
                    SessionResult(SessionResult.RESULT_SUCCESS)
                } catch (e: Exception) {
                    Timber.e(e, "Error executing playItemsUseCase from custom command.")
                    SessionResult(SessionError.ERROR_UNKNOWN)
                }
            }
        }
        return Futures.immediateFuture(SessionResult(SessionError.ERROR_NOT_SUPPORTED))
    }
}

override fun onTaskRemoved(rootIntent: Intent?) {
    if (!player.playWhenReady && player.playbackState != Player.STATE_BUFFERING) {
        stopSelf()
    }
}

override fun onDestroy() {
    player.removeListener(playerListener)
    if (::notificationManager.isInitialized) {
        notificationManager.setPlayer(null)
```

```
        }
        mediaSession?.release()
        mediaSession = null
        serviceScope.cancel()
        super.onDestroy()
    }
}

// File: java\com\example\holodex\ui\MainActivity.kt
// File: java/com/example/holodex/ui/MainActivity.kt
@file:OptIn(ExperimentalMaterial3Api::class)

@file:Suppress("OPT_IN_USAGE") // Optional: To silence the false warning for UnstableApi
@file:androidx.media3.common.util.UnstableApi

package com.example.holodex.ui

import android.Manifest
import android.annotation.SuppressLint
import android.content.ComponentName
import android.content.Intent
import android.content.pm.PackageManager
import android.os.Build
import android.os.Bundle
import android.widget.Toast
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.result.contract.ActivityResultContracts
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.WindowInsets
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.safeDrawing
import androidx.compose.foundation.layout.windowInsetsPadding
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material3.Button
import androidx.compose.material3.ExperimentalMaterial3Api
import androidx.compose.material3.Icon
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.ModalBottomSheet
import androidx.compose.material3.NavigationBar
import androidx.compose.material3.NavigationBarItem
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Text
import androidx.compose.material3.rememberModalBottomSheetState
import androidx.compose.runtime.Composable
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.rememberCoroutineScope
import androidx.compose.runtime.setValue
```

```
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.text.style.TextAlign
import androidx.compose.ui.unit.dp
import androidx.core.content.ContextCompat
import androidx.core.net.toUri
import androidx.core.splashscreen.SplashScreen.Companion.installSplashScreen
import androidx.core.view.WindowCompat
import androidx.hilt.navigation.compose.hiltViewModel
import androidx.lifecycle.compose.collectAsStateWithLifecycle
import androidx.media3.common.Player
import androidx.media3.session.MediaController
import androidx.media3.session.SessionCommand
import androidx.media3.session.SessionResult
import androidx.media3.session.SessionToken
import androidx.navigation.NavController
import androidx.navigation.NavGraph.Companion.findStartDestination
import androidx.navigation.NavHostController
import androidx.navigation.NavType
import androidx.navigation.compose.NavHost
import androidx.navigation.compose.composable
import androidx.navigation.compose.currentBackStackEntryAsState
import androidx.navigation.compose.rememberNavController
import androidx.navigation.navArgument
import com.example.holodex.MyApp
import com.example.holodex.R
import com.example.holodex.auth.LoginScreen
import com.example.holodex.data.db.LikedItemType
import com.example.holodex.playback.PlaybackRequestManager
import com.example.holodex.playback.domain.model.PlaybackItem
import com.example.holodex.service.ARG_PLAYBACK_ITEMS_LIST
import com.example.holodex.service.ARG_SHOULD_SHUFFLE
import com.example.holodex.service.ARG_START_INDEX
import com.example.holodex.service.ARG_START_POSITION_SEC
import com.example.holodex.service.CUSTOM_COMMAND_PREPARE_FROM_REQUEST
import com.example.holodex.service.MediaPlaybackService
import com.example.holodex.ui.composables.FullPlayerActions
import com.example.holodex.ui.composables.FullPlayerScreenContent
import com.example.holodex.ui.composables.MiniPlayerWithProgressBar
import com.example.holodex.ui.dialogs.CreatePlaylistDialog
import com.example.holodex.ui.dialogs.SelectPlaylistDialog
import com.example.holodex.ui.screens.ChannelScreen
import com.example.holodex.ui.screens.DiscoveryScreen
import com.example.holodex.ui.screens.DownloadsScreen
import com.example.holodex.ui.screens.ExternalChannelScreen
import com.example.holodex.ui.screens.ForYouScreen
import com.example.holodex.ui.screens.FullListViewScreen
import com.example.holodex.ui.screens.HomeScreen
import com.example.holodex.ui.screens.LibraryScreen
import com.example.holodex.ui.screens.PlaylistDetailsScreen
import com.example.holodex.ui.screens.SettingsScreen
import com.example.holodex.ui.screens.VideoDetailsScreen
import com.example.holodex.ui.screens.navigation.BottomNavItem
```

```
import com.example.holodex.ui.theme.HolodexMusicTheme
import com.example.holodex.viewmodel.ChannelDetailsViewModel
import com.example.holodex.viewmodel.FavoritesViewModel
import com.example.holodex.viewmodel.FullListViewModel
import com.example.holodex.viewmodel.PlaybackViewModel
import com.example.holodex.viewmodel.PlaylistDetailsViewModel
import com.example.holodex.viewmodel.PlaylistManagementViewModel
import com.example.holodex.viewmodel.SettingsViewModel
import com.example.holodex.viewmodel.VideoDetailsViewModel
import com.example.holodex.viewmodel.VideoListViewModel
import com.example.holodex.viewmodel.VideoListViewModel.MusicCategoryType
import com.google.common.util.concurrent.ListenableFuture
import com.google.common.util.concurrent.MoreExecutors
import dagger.hilt.android.AndroidEntryPoint
import kotlinx.coroutines.flow.collectLatest
import kotlinx.coroutines.launch
import timber.log.Timber
import java.net.URLEncoder
import java.nio.charset.StandardCharsets
import javax.inject.Inject

object AppDestinations {
    const val HOME_ROUTE = "home"
    const val DISCOVERY_ROUTE = "discover"
    const val LIBRARY_ROUTE = "library"
    const val DOWNLOADS_ROUTE = "downloads"
    const val SETTINGS_ROUTE = "settings"
    const val FULL_PLAYER_ROUTE = "full_player_dialog"

    const val VIDEO_DETAILS_ROUTE_TEMPLATE = "video_details/{${VideoDetailsViewModel.VIDEO_ID_
    fun videoDetailRoute(videoId: String) = "video_details/$videoId"

    const val LOGIN_ROUTE = "login"

    const val FULL_LIST_VIEW_ROUTE_TEMPLATE =
        "full_list/{${FullListViewModel.CATEGORY_TYPE_ARG}}/{${FullListViewModel.ORG_ARG}}"

    fun fullListViewRoute(category: MusicCategoryType, org: String): String {
        val encodedOrg = URLEncoder.encode(org, StandardCharsets.UTF_8.toString())
        return "full_list/${category.name}/$encodedOrg"
    }

    const val FOR_YOU_ROUTE = "for_you"

    const val PLAYLIST_DETAILS_ROUTE_TEMPLATE =
        "playlist_details/{${PlaylistDetailsViewModel.PLAYLIST_ID_ARG}}"

    fun playlistDetailsRoute(playlistId: String): String {
        val encodedId = URLEncoder.encode(playlistId, StandardCharsets.UTF_8.toString())
        return "playlist_details/$encodedId"
    }
}

@AndroidEntryPoint
class MainActivity : ComponentActivity() {
```

```
@Inject
lateinit var playbackRequestManager: PlaybackRequestManager

@Inject
lateinit var player: Player

private val myApp: MyApp by lazy { application as MyApp }
private var mediaController: MediaController? = null
private lateinit var sessionToken: SessionToken

companion object {
    private const val TAG = "MainActivity"
}

private val requestPermissionLauncher =
    registerForActivityResult(ActivityResultContracts.RequestMultiplePermissions()) { perm
        permissions.entries.forEach {
            Timber.d("Permission [${it.key}] granted: ${it.value}")
        }
    }

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    WindowCompat.setDecorFitsSystemWindows(window, false)
    installSplashScreen()
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU && ContextCompat.checkSelfPe
            this,
            Manifest.permission.POST_NOTIFICATIONS
        ) != PackageManager.PERMISSION_GRANTED
    ) {
        requestPermissions(arrayOf(Manifest.permission.POST_NOTIFICATIONS), 101)
    }
    sessionToken = SessionToken(this, ComponentName(this, MediaPlaybackService::class.java
    checkAndRequestPermissions()

    setContent {
        val navController = rememberNavController()
        HolodexApp(
            navController = navController, // Pass the controller
            playbackRequestManager = playbackRequestManager,
            activity = this,
            player = player

        )
    }
}

private fun checkAndRequestPermissions() {
    val permissionsToRequest = mutableListOf<String>()

    // Notification permission for Android 13+
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
        if (ContextCompat.checkSelfPermission(
                this,
```

```
                Manifest.permission.POST_NOTIFICATIONS
            ) != PackageManager.PERMISSION_GRANTED
        ) {
            permissionsToRequest.add(Manifest.permission.POST_NOTIFICATIONS)
        }
    }

    // Storage permission for Android 9 and below
    if (Build.VERSION.SDK_INT <= Build.VERSION_CODES.P) {
        if (ContextCompat.checkSelfPermission(
                this,
                Manifest.permission.WRITE_EXTERNAL_STORAGE
            ) != PackageManager.PERMISSION_GRANTED
        ) {
            permissionsToRequest.add(Manifest.permission.WRITE_EXTERNAL_STORAGE)
        }
    }

    if (permissionsToRequest.isNotEmpty()) {
        requestPermissionLauncher.launch(permissionsToRequest.toTypedArray())
    }
}

// ... (onStart, onStop, sendPlaybackRequestToService, etc. remain the same)
override fun onStart() {
    super.onStart()
    connectMediaController()

    // NEW: Trigger reconciliation whenever the app becomes visible
    Timber.d("MainActivity onStart: Triggering download reconciliation.")
    myApp.reconcileCompletedDownloads()
}

override fun onStop() {
    super.onStop()
    // Don't release MediaController in onStop - only disconnect if needed
    // The MediaController should persist across onStop/onStart cycles
    Timber.tag(TAG).d("MainActivity onStop - keeping MediaController connected")
}

override fun onDestroy() {
    super.onDestroy()
    // Only release MediaController when the activity is actually being destroyed
    mediaController?.release()
    mediaController = null
    Timber.tag(TAG).d("MediaController released in onDestroy")
}

internal fun sendPlaybackRequestToService(
    items: List<PlaybackItem>,
    startIndex: Int,
    startPositionSec: Long,
    shouldShuffle: Boolean = false
) {
    if (items.isEmpty()) {
```

```
    Timber.tag(TAG)
        .w("sendPlaybackRequestToService called with empty item list. Aborting."); ret
}
val serviceIntent = Intent(this, MediaPlaybackService::class.java)
try {
    items.forEach { item ->
        if (item.streamUri?.startsWith("content://") == true) {
            val uri = item.streamUri!!.toUri()
            grantUriPermission(
                "com.example.holodex",
                uri,
                Intent.FLAG_GRANT_READ_URI_PERMISSION
            )
            Timber.d("Granted READ permission for URI: $uri")
        }
    }
} catch (e: Exception) {
    Timber.e(e, "Failed to grant URI permissions.")
    // Optionally show a toast if permission granting fails, as playback will likely f
}
// --- END OF FIX ---

startService(serviceIntent)
Timber.tag(TAG).d("MediaPlaybackService explicitly started/ensured running.")
if (mediaController == null || !mediaController!!.isConnected) {
    Timber.tag(TAG)
        .w("MediaController not available/connected. Attempting to connect then send."
    connectMediaController { success ->
        if (success) {
            sendPlaybackRequestToService(
                items,
                startIndex,
                startPositionSec,
                shouldShuffle
            )
        } else {
            Toast.makeText(
                this,
                getString(R.string.error_player_service_not_ready),
                Toast.LENGTH_SHORT
            ).show()
        }
    }; return
}
val commandArgs = Bundle().apply {
    putParcelableArrayList(ARG_PLAYBACK_ITEMS_LIST, ArrayList(items))
    putInt(ARG_START_INDEX, startIndex)
    putLong(ARG_START_POSITION_SEC, startPositionSec)
    putBoolean(ARG_SHOULD_SHUFFLE, shouldShuffle)
}
val command = SessionCommand(CUSTOM_COMMAND_PREPARE_FROM_REQUEST, Bundle.EMPTY)
Timber.tag(TAG)
    .d("Sending $CUSTOM_COMMAND_PREPARE_FROM_REQUEST with ${items.size} items, startIn
val resultFuture: ListenableFuture<SessionResult> =
    mediaController!!.sendCustomCommand(command, commandArgs)
```

```
    resultFuture.addListener({
        try {
            val result: SessionResult = resultFuture.get()
            if (result.resultCode != SessionResult.RESULT_SUCCESS) {
                Timber.tag(TAG)
                    .w("Custom command $CUSTOM_COMMAND_PREPARE_FROM_REQUEST failed: ${resu
                Toast.makeText(
                    this,
                    getString(R.string.error_playback_command_failed, result.resultCode),
                    Toast.LENGTH_LONG
                ).show()
            } else {
                Timber.tag(TAG)
                    .i("Custom command $CUSTOM_COMMAND_PREPARE_FROM_REQUEST successful.")
            }
        } catch (e: Exception) {
            Timber.tag(TAG)
                .e(e, "Error processing result of $CUSTOM_COMMAND_PREPARE_FROM_REQUEST")
            Toast.makeText(
                this,
                getString(R.string.error_initiating_playback),
                Toast.LENGTH_LONG
            ).show()
        }
    }, MoreExecutors.directExecutor())
}

private fun connectMediaController(onConnected: ((Boolean) -> Unit)? = null) {
    if (mediaController?.isConnected == true) {
        Timber.tag(TAG).d("MediaController already connected.")
        onConnected?.invoke(true)
        return
    }

    // Clean up any existing controller that might be in a bad state
    if (mediaController != null && !mediaController!!.isConnected) {
        Timber.tag(TAG).d("Cleaning up disconnected MediaController")
        try {
            mediaController?.release()
        } catch (e: Exception) {
            Timber.tag(TAG).w(e, "Error releasing old MediaController")
        }
        mediaController = null
    }

    Timber.tag(TAG).d("Attempting to connect MediaController.")

    // Ensure service is started before attempting connection
    val serviceIntent = Intent(this, MediaPlaybackService::class.java)
    startService(serviceIntent)

    val controllerFuture: ListenableFuture<MediaController> =
        MediaController.Builder(this, sessionToken).buildAsync()

    controllerFuture.addListener({
```

```
            try {
                val controller = controllerFuture.get()
                mediaController = controller
                Timber.tag(TAG).d("MediaController connected successfully")
                onConnected?.invoke(true)
            } catch (e: Exception) {
                mediaController = null
                Timber.tag(TAG).e(e, "Error connecting MediaController")
                onConnected?.invoke(false)
            }
        }, MoreExecutors.directExecutor())
    }
}

@SuppressLint("UnstableApi")
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun HolodexApp(
    navController: NavHostController,
    playbackRequestManager: PlaybackRequestManager,
    activity: ComponentActivity,
    player: Player?
) {
    val coroutineScope = rememberCoroutineScope()
    var showFullPlayerSheet by remember { mutableStateOf(false) }
    val fullPlayerSheetState = rememberModalBottomSheetState(skipPartiallyExpanded = true)

    // ViewModels for globally-scoped UI state (theme, player, dialogs) are hoisted here.
    val settingsViewModel: SettingsViewModel = hiltViewModel()
    val playbackViewModel: PlaybackViewModel = hiltViewModel()
    val playlistManagementViewModel: PlaylistManagementViewModel = hiltViewModel(activity)
    val videoListViewModel: VideoListViewModel = hiltViewModel(activity)

    val playbackUiState by playbackViewModel.uiState.collectAsStateWithLifecycle()
    val isPlayerActive = playbackUiState.currentItem != null
    val navBackStackEntry by navController.currentBackStackEntryAsState()
    val currentRoute = navBackStackEntry?.destination?.route

    val navigationRequest by videoListViewModel.navigationRequest.collectAsStateWithLifecycle(

    LaunchedEffect(navigationRequest) {
        navigationRequest?.let { destination ->
            // --- LOGGING POINT: This should now appear ---
            Timber.d("HolodexApp: NavigationRequest observed: $destination")
            when (destination) {
                is VideoListViewModel.NavigationDestination.VideoDetails -> {
                    navController.navigate(AppDestinations.videoDetailRoute(destination.videoI
                }

                is VideoListViewModel.NavigationDestination.HomeScreenWithSearch -> {
                    // This can be used to navigate home and pop up the search bar
                    navController.navigate(AppDestinations.HOME_ROUTE) {
                        popUpTo(navController.graph.startDestinationId) { saveState = true }
                        launchSingleTop = true
                    }
```

```
                // A mechanism to activate search would be needed here, e.g., via the View
                videoListViewModel.setSearchActive(true)
            }
        }
        // Important: Consume the navigation event so it doesn't trigger again on recompos
        videoListViewModel.clearNavigationRequest()
    }
}

LaunchedEffect(playbackRequestManager) {
    playbackRequestManager.playbackRequest.collectLatest { request ->
        (activity as MainActivity).sendPlaybackRequestToService(
            request.items, request.startIndex, request.startPositionSec, request.shouldShu
        )
    }
}

HolodexMusicTheme(settingsViewModel = settingsViewModel) {
    Scaffold(
        modifier = Modifier
            .fillMaxSize()
            .windowInsetsPadding(WindowInsets.safeDrawing),
        bottomBar = {
            Column {
                if (isPlayerActive) {
                    MiniPlayerWithProgressBar(
                        playbackViewModel = playbackViewModel,
                        onTapped = { showFullPlayerSheet = true }
                    )
                }
                val navItems = listOf(
                    BottomNavItem.Discover,
                    BottomNavItem.Browse,
                    BottomNavItem.Library,
                    BottomNavItem.Downloads
                )
                NavigationBar {
                    navItems.forEach { item ->
                        val isSelected = currentRoute == item.route
                        NavigationBarItem(
                            icon = {
                                Icon(
                                    item.icon,
                                    contentDescription = stringResource(item.titleResId)
                                )
                            },
                            label = { Text(stringResource(item.titleResId)) },
                            selected = isSelected,
                            onClick = {
                                // SPECIAL HANDLING for start destination
                                if (item.route == AppDestinations.DISCOVERY_ROUTE &&
                                    navController.graph.startDestinationId == navControlle
                                        AppDestinations.DISCOVERY_ROUTE
                                    )?.id
                                ) {
```

```
                                // Force navigation to start destination by clearing s
                                navController.navigate(item.route) {
                                    popUpTo(0) { inclusive = true }
                                    launchSingleTop = true
                                }
                            } else {
                                // Normal navigation for other destinations
                                navController.navigate(item.route) {
                                    popUpTo(navController.graph.findStartDestination()
                                        saveState = true
                                    }
                                    launchSingleTop = true
                                    restoreState = true
                                }
                            }
                        }
                    )
                }
            }
        }
    ) { innerPadding ->
        NavHost(
            navController = navController,
            startDestination = AppDestinations.LIBRARY_ROUTE,
            modifier = Modifier
                .padding(innerPadding)
                .fillMaxSize()
        ) {
            composable(AppDestinations.DISCOVERY_ROUTE) { DiscoveryScreen(navController =
            composable(AppDestinations.FOR_YOU_ROUTE) { ForYouScreen(navController = navCo
            composable(AppDestinations.HOME_ROUTE) {
                val currentApiKey by settingsViewModel.currentApiKey.collectAsStateWithLif
                if (currentApiKey.isBlank()) {
                    ApiKeyMissingContent(navController = navController)
                } else {
                    HomeScreen(
                        navController = navController,
                        videoListViewModel = videoListViewModel,
                        playlistManagementViewModel = playlistManagementViewModel // <-- P
                    )
                }
            }
            composable(
                route = "external_channel_details/{${ChannelDetailsViewModel.CHANNEL_ID_AR
                arguments = listOf(navArgument(ChannelDetailsViewModel.CHANNEL_ID_ARG) {
                    type = NavType.StringType
                })
            ) {
                ExternalChannelScreen(
                    navController = navController,
                    onNavigateUp = { navController.popBackStack() }
                )
            }
```

```
composable(
    route = "channel_details/{${ChannelDetailsViewModel.CHANNEL_ID_ARG}}",
    arguments = listOf(navArgument(ChannelDetailsViewModel.CHANNEL_ID_ARG) {
        type = NavType.StringType
    })
) {
    ChannelScreen(
        navController = navController,
        onNavigateUp = { navController.popBackStack() }
    )
}
composable(AppDestinations.LIBRARY_ROUTE) {
    LibraryScreen(
        navController = navController,
        playlistManagementViewModel = playlistManagementViewModel // <-- PASS
    )
}
composable(AppDestinations.DOWNLOADS_ROUTE) {
    DownloadsScreen(
        navController = navController,
        playlistManagementViewModel = playlistManagementViewModel // <-- PASS
    )
}
composable(AppDestinations.SETTINGS_ROUTE) {
    val videoListViewModel: VideoListViewModel = hiltViewModel(activity)
    SettingsScreen(
        navController = navController,
        onNavigateUp = { navController.popBackStack() },
        onApiKeySavedRestartNeeded = { videoListViewModel.refreshCurrentListVi
    )
}
composable(AppDestinations.LOGIN_ROUTE) { LoginScreen(onLoginSuccess = { navCo
composable(
    route = AppDestinations.FULL_LIST_VIEW_ROUTE_TEMPLATE,
    arguments = listOf(
        navArgument(FullListViewModel.CATEGORY_TYPE_ARG) {
            type = NavType.StringType
        },
        navArgument(FullListViewModel.ORG_ARG) { type = NavType.StringType }
    )
) {
    FullListViewScreen(
        navController = navController,
        categoryType = MusicCategoryType.valueOf(
            it.arguments?.getString(
                FullListViewModel.CATEGORY_TYPE_ARG
            ) ?: MusicCategoryType.TRENDING.name
        )
    )
}
composable(
    AppDestinations.PLAYLIST_DETAILS_ROUTE_TEMPLATE,
    arguments = listOf(navArgument(PlaylistDetailsViewModel.PLAYLIST_ID_ARG) {
        type = NavType.StringType
    })
```

```
            ) {
                PlaylistDetailsScreen(
                    navController = navController,
                    playlistManagementViewModel = playlistManagementViewModel, // <-- PASS
                    onNavigateUp = { navController.popBackStack() })
            }
            composable(
                AppDestinations.VIDEO_DETAILS_ROUTE_TEMPLATE,
                arguments = listOf(navArgument(VideoDetailsViewModel.VIDEO_ID_ARG) {
                    type = NavType.StringType
                })
            ) {
                VideoDetailsScreen(
                    navController = navController,
                    onNavigateUp = { navController.popBackStack() })
            }
        }
    }
}

if (showFullPlayerSheet) {
    ModalBottomSheet(
        onDismissRequest = { showFullPlayerSheet = false },
        sheetState = fullPlayerSheetState,
        containerColor = Color.Transparent,
        shape = RoundedCornerShape(0),
        scrimColor = Color.Transparent,
        dragHandle = null
    ) {
        FullPlayerScreenDestination(
            player = player,
            navController = navController,
            onNavigateUp = {
                coroutineScope.launch { fullPlayerSheetState.hide() }.invokeOnCompletion {
                    if (!fullPlayerSheetState.isVisible) {
                        showFullPlayerSheet = false
                    }
                }
            }
        )
    }
}

val showSelectPlaylistDialog by playlistManagementViewModel.showSelectPlaylistDialog.colle
val userPlaylistsForDialog by playlistManagementViewModel.userPlaylists.collectAsStateWith
val showCreatePlaylistDialog by playlistManagementViewModel.showCreatePlaylistDialog.colle

if (showSelectPlaylistDialog) {
    SelectPlaylistDialog(
        playlists = userPlaylistsForDialog,
        onDismissRequest = { playlistManagementViewModel.cancelAddToPlaylistFlow() },
        onPlaylistSelected = { playlist ->
            playlistManagementViewModel.addItemToExistingPlaylist(
                playlist
            )
```

```
            },
            onCreateNewPlaylistClicked = { playlistManagementViewModel.handleCreateNewPlaylist
        )
    }
    if (showCreatePlaylistDialog) {
        CreatePlaylistDialog(
            onDismissRequest = { playlistManagementViewModel.cancelAddToPlaylistFlow() },
            onCreatePlaylist = { name, desc ->
                playlistManagementViewModel.confirmCreatePlaylist(
                    name,
                    desc
                )
            }
        )
    }
}

@Composable
private fun FullPlayerScreenDestination(
    player: Player?,
    navController: NavHostController, // FIX: Accept navController
    onNavigateUp: () -> Unit,
    modifier: Modifier = Modifier
) {
    val playbackViewModel: PlaybackViewModel = hiltViewModel()
    val favoritesViewModel: FavoritesViewModel = hiltViewModel()
    val playlistManagementViewModel: PlaylistManagementViewModel = hiltViewModel()
    val videoListViewModel: VideoListViewModel = hiltViewModel()

    val playerActions = remember(
        playbackViewModel,
        favoritesViewModel,
        videoListViewModel,
        playlistManagementViewModel
    ) {
        FullPlayerActions(
            onNavigateUp = onNavigateUp,
            onTogglePlayPause = { playbackViewModel.togglePlayPause() },
            onSeekTo = { positionSec -> playbackViewModel.seekTo(positionSec) },
            onSkipToNext = { playbackViewModel.skipToNext() },
            onSkipToPrevious = { playbackViewModel.skipToPrevious() },
            onToggleRepeatMode = { playbackViewModel.toggleRepeatMode() },
            onToggleShuffleMode = { playbackViewModel.toggleShuffleMode() },
            onPlayQueueItemAtIndex = { index -> playbackViewModel.playQueueItemAtIndex(index)
            onReorderQueueItem = { from, to -> playbackViewModel.reorderQueueItem(from, to) },
            onRemoveQueueItem = { index -> playbackViewModel.removeItemFromQueue(index) },
            onClearQueue = { playbackViewModel.clearCurrentQueue() },
            onToggleLike = { playbackItem -> favoritesViewModel.toggleLike(playbackItem) },
            onFindArtist = { channelId -> videoListViewModel.performSearchByChannelId(channelI
            onOpenAudioSettings = { audioSessionId -> Timber.d("Action: Open Audio Settings fo
            onAddToPlaylist = { playbackItem ->

                playlistManagementViewModel.prepareItemForPlaylistAdditionFromPlaybackItem(pla
            }
        )
```

```
    }

    FullPlayerScreenContent(
        player = player,
        navController = navController,
        actions = playerActions,
        modifier = modifier
    )
}

@Composable
private fun ApiKeyMissingContent(
    navController: NavController,
) {
    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        Text(
            stringResource(R.string.status_api_key_required_main),
            style = MaterialTheme.typography.bodyLarge,
            textAlign = TextAlign.Center
        )
        Spacer(Modifier.height(16.dp))
        Button(onClick = {
            navController.navigate(AppDestinations.SETTINGS_ROUTE) {
                launchSingleTop = true
            }
        }) { Text(stringResource(R.string.button_go_to_settings)) }
    }
}

// File: java\com\example\holodex\ui\MainScreenScaffold.kt
package com.example.holodex.ui

import android.annotation.SuppressLint
import android.util.Log
import android.widget.Toast
import androidx.activity.ComponentActivity
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material3.ExperimentalMaterial3Api
import androidx.compose.material3.Icon
import androidx.compose.material3.ModalBottomSheet
import androidx.compose.material3.NavigationBar
import androidx.compose.material3.NavigationBarItem
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Text
import androidx.compose.material3.rememberModalBottomSheetState
```

```
import androidx.compose.runtime.Composable
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.rememberCoroutineScope
import androidx.compose.runtime.setValue
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.res.stringResource
import androidx.hilt.navigation.compose.hiltViewModel
import androidx.media3.common.Player
import androidx.navigation.NavGraph.Companion.findStartDestination
import androidx.navigation.NavHostController
import androidx.navigation.compose.currentBackStackEntryAsState
import com.example.holodex.playback.PlaybackRequestManager
import com.example.holodex.ui.composables.FullPlayerActions
import com.example.holodex.ui.composables.FullPlayerScreenContent
import com.example.holodex.ui.composables.MiniPlayerWithProgressBar
import com.example.holodex.ui.composables.PlaylistManagementDialogs
import com.example.holodex.ui.navigation.AppDestinations
import com.example.holodex.ui.navigation.HolodexNavHost
import com.example.holodex.ui.screens.navigation.BottomNavItem
import com.example.holodex.ui.theme.HolodexMusicTheme
import com.example.holodex.viewmodel.FavoritesViewModel
import com.example.holodex.viewmodel.PlaybackViewModel
import com.example.holodex.viewmodel.PlaylistManagementViewModel
import com.example.holodex.viewmodel.SettingsViewModel
import com.example.holodex.viewmodel.VideoListSideEffect
import com.example.holodex.viewmodel.VideoListViewModel
import kotlinx.coroutines.flow.collectLatest
import kotlinx.coroutines.launch
import org.orbitmvi.orbit.compose.collectSideEffect

private const val TAG = "MainScreenScaffold"

@SuppressLint("UnstableApi")
@androidx.annotation.OptIn(androidx.media3.common.util.UnstableApi::class)
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun MainScreenScaffold(
    navController: NavHostController,
    playbackRequestManager: PlaybackRequestManager,
    activity: ComponentActivity,
    player: Player
) {
    Log.d(TAG, "MainScreenScaffold: Composing")

    val coroutineScope = rememberCoroutineScope()
    val context = LocalContext.current

    // ViewModels
    val settingsViewModel: SettingsViewModel = hiltViewModel()
    val playbackViewModel: PlaybackViewModel = hiltViewModel()
```

```
val playlistManagementViewModel: PlaylistManagementViewModel = hiltViewModel(activity)
val videoListViewModel: VideoListViewModel = hiltViewModel(activity)

Log.d(TAG, "MainScreenScaffold: ViewModels created")

// UI State
var showFullPlayerSheet by remember { mutableStateOf(false) }
val fullPlayerSheetState = rememberModalBottomSheetState(skipPartiallyExpanded = true)

// Navigation State
val navBackStackEntry by navController.currentBackStackEntryAsState()
val currentRoute = navBackStackEntry?.destination?.route

// --- Orbit Side Effects ---
videoListViewModel.collectSideEffect { sideEffect ->
    when (sideEffect) {
        is VideoListSideEffect.NavigateTo -> {
            when (val destination = sideEffect.destination) {
                is VideoListViewModel.NavigationDestination.VideoDetails -> {
                    navController.navigate(AppDestinations.videoDetailRoute(destination.vi
                }
                is VideoListViewModel.NavigationDestination.HomeScreenWithSearch -> {
                    navController.navigate(AppDestinations.HOME_ROUTE) {
                        popUpTo(navController.graph.startDestinationId) { saveState = true
                        launchSingleTop = true
                    }
                    videoListViewModel.setSearchActive(true)
                }
            }
        }
        is VideoListSideEffect.ShowToast -> {
            Toast.makeText(context, sideEffect.message, Toast.LENGTH_SHORT).show()
        }
    }
}

// Bridge Playback
LaunchedEffect(playbackRequestManager) {
    playbackRequestManager.playbackRequest.collectLatest { request ->
        (activity as MainActivity).sendPlaybackRequestToService(
            request.items, request.startIndex, request.startPositionSec, request.shouldShu
        )
    }
}

HolodexMusicTheme(settingsViewModel = settingsViewModel) {
    Scaffold(
        modifier = Modifier.fillMaxSize(),
        bottomBar = {
            Log.d(TAG, "bottomBar: Composing")
            Column {
                Log.d(TAG, "bottomBar Column: About to compose MiniPlayer")
                MiniPlayerWithProgressBar(
                    playbackViewModel = playbackViewModel,
                    onTapped = { showFullPlayerSheet = true }
```

```
            )
            Log.d(TAG, "bottomBar Column: MiniPlayer composed, now composing Navigatio

            NavigationBar {
                val navItems = listOf(
                    BottomNavItem.Discover,
                    BottomNavItem.Browse,
                    BottomNavItem.Library,
                    BottomNavItem.Downloads
                )
                navItems.forEach { item ->
                    val isSelected = currentRoute == item.route
                    NavigationBarItem(
                        icon = { Icon(item.icon, contentDescription = null) },
                        label = { Text(stringResource(item.titleResId)) },
                        selected = isSelected,
                        onClick = {
                            if (item.route == AppDestinations.DISCOVERY_ROUTE &&
                                navController.graph.startDestinationId == navControlle
                                navController.navigate(item.route) {
                                    popUpTo(0) { inclusive = true }
                                    launchSingleTop = true
                                }
                            } else {
                                navController.navigate(item.route) {
                                    popUpTo(navController.graph.findStartDestination()
                                    launchSingleTop = true
                                    restoreState = true
                                }
                            }
                        }
                    )
                }
            }
            Log.d(TAG, "bottomBar Column: NavigationBar composed")
        }
    }
) { innerPadding ->
    Box(modifier = Modifier.padding(innerPadding)) {
        HolodexNavHost(
            navController = navController,
            videoListViewModel = videoListViewModel,
            playlistManagementViewModel = playlistManagementViewModel,
            activity = activity
        )
    }
}

if (showFullPlayerSheet) {
    ModalBottomSheet(
        onDismissRequest = { showFullPlayerSheet = false },
        sheetState = fullPlayerSheetState,
        containerColor = Color.Transparent,
        shape = RoundedCornerShape(0),
        scrimColor = Color.Black.copy(alpha = 0.6f),
```

```
                dragHandle = null
            ) {
                FullPlayerScreenDestination(
                    player = player,
                    navController = navController,
                    onNavigateUp = {
                        coroutineScope.launch { fullPlayerSheetState.hide() }.invokeOnCompleti
                            if (!fullPlayerSheetState.isVisible) showFullPlayerSheet = false
                        }
                    }
                )
            }
        }

        PlaylistManagementDialogs(playlistManagementViewModel)
    }
}

@androidx.annotation.OptIn(androidx.media3.common.util.UnstableApi::class)
@Composable
private fun FullPlayerScreenDestination(
    player: Player?,
    navController: NavHostController,
    onNavigateUp: () -> Unit,
    modifier: Modifier = Modifier
) {
    val playbackViewModel: PlaybackViewModel = hiltViewModel()
    val favoritesViewModel: FavoritesViewModel = hiltViewModel()
    val playlistManagementViewModel: PlaylistManagementViewModel = hiltViewModel()
    val videoListViewModel: VideoListViewModel = hiltViewModel()
    val context = LocalContext.current

    val playerActions = remember(playbackViewModel, favoritesViewModel, videoListViewModel, pl
        FullPlayerActions(
            onNavigateUp = onNavigateUp,
            onTogglePlayPause = { playbackViewModel.togglePlayPause() },
            onSeekTo = { positionSec -> playbackViewModel.seekTo(positionSec) },
            onSkipToNext = { playbackViewModel.skipToNext() },
            onSkipToPrevious = { playbackViewModel.skipToPrevious() },
            onToggleRepeatMode = { playbackViewModel.toggleRepeatMode() },
            onToggleShuffleMode = { playbackViewModel.toggleShuffleMode() },
            onPlayQueueItemAtIndex = { index -> playbackViewModel.playQueueItemAtIndex(index)
            onReorderQueueItem = { from, to -> playbackViewModel.reorderQueueItem(from, to) },
            onRemoveQueueItem = { index -> playbackViewModel.removeItemFromQueue(index) },
            onClearQueue = { playbackViewModel.clearCurrentQueue() },
            onToggleLike = { playbackItem -> favoritesViewModel.toggleLike(playbackItem) },
            onFindArtist = { channelId -> videoListViewModel.setBrowseContextAndNavigate(chann
            onOpenAudioSettings = { audioSessionId ->
                Toast.makeText(context, "Audio FX Session ID: $audioSessionId", Toast.LENGTH_S
            },
            onAddToPlaylist = { playbackItem ->
                playlistManagementViewModel.prepareItemForPlaylistAdditionFromPlaybackItem(pla
            }
        )
    }
```

```
    FullPlayerScreenContent(
        player = player,
        navController = navController,
        actions = playerActions,
        modifier = modifier
    )
}

// File: java\com\example\holodex\ui\composables\ApiKeyInputScreen.kt
package com.example.holodex.ui.composables

import android.widget.Toast
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.text.KeyboardActions
import androidx.compose.foundation.text.KeyboardOptions
import androidx.compose.material3.Button
import androidx.compose.material3.OutlinedTextField
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.platform.LocalFocusManager
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.text.input.ImeAction
import androidx.compose.ui.text.input.KeyboardType
import androidx.compose.ui.text.input.PasswordVisualTransformation
import androidx.compose.ui.unit.dp
import androidx.hilt.navigation.compose.hiltViewModel
import androidx.lifecycle.compose.collectAsStateWithLifecycle
import com.example.holodex.R
import com.example.holodex.viewmodel.ApiKeySaveResult
import com.example.holodex.viewmodel.SettingsViewModel

@Composable
fun ApiKeyInputScreen(
    settingsViewModel: SettingsViewModel = hiltViewModel(),
    onApiKeySavedSuccessfully: () -> Unit,
    modifier: Modifier = Modifier,
) {
    val currentApiKey by settingsViewModel.currentApiKey.collectAsStateWithLifecycle()
    var apiKeyInputText by remember(currentApiKey) { mutableStateOf(currentApiKey) } // Initia

    val context = LocalContext.current
    val focusManager = LocalFocusManager.current
    val apiKeySaveResult by settingsViewModel.apiKeySaveResult.collectAsStateWithLifecycle()
```

```
    LaunchedEffect(apiKeySaveResult) {
        when (val result = apiKeySaveResult) {
            is ApiKeySaveResult.Success -> {
                Toast.makeText(context, R.string.toast_api_key_saved, Toast.LENGTH_SHORT).show
                onApiKeySavedSuccessfully()
                settingsViewModel.resetApiKeySaveResult()
            }
            is ApiKeySaveResult.Empty -> {
                Toast.makeText(context, R.string.toast_api_key_empty, Toast.LENGTH_SHORT).show
                settingsViewModel.resetApiKeySaveResult()
            }
            is ApiKeySaveResult.Error -> {
                Toast.makeText(context, result.message, Toast.LENGTH_LONG).show()
                settingsViewModel.resetApiKeySaveResult()
            }
            is ApiKeySaveResult.Idle -> { /* Do nothing */ }
        }
    }

    Column(
        modifier = modifier.padding(bottom = 8.dp), // Add some bottom padding
        verticalArrangement = Arrangement.spacedBy(8.dp)
    ) {
        OutlinedTextField(
            value = apiKeyInputText,
            onValueChange = { apiKeyInputText = it },
            label = { Text(stringResource(id = R.string.hint_api_key)) },
            modifier = Modifier.fillMaxWidth(),
            singleLine = true,
            visualTransformation = PasswordVisualTransformation(),
            keyboardOptions = KeyboardOptions.Default.copy(
                keyboardType = KeyboardType.Password,
                imeAction = ImeAction.Done
            ),
            keyboardActions = KeyboardActions(onDone = {
                focusManager.clearFocus()
                settingsViewModel.saveApiKey(apiKeyInputText)
            })
        )
        Button(
            onClick = {
                focusManager.clearFocus()
                settingsViewModel.saveApiKey(apiKeyInputText)
            },
            modifier = Modifier.align(Alignment.End)
        ) {
            Text(stringResource(id = R.string.button_save_key))
        }
    }
}

// File: java\com\example\holodex\ui\composables\CarouselShelf.kt
// File: java/com/example/holodex/ui/composables/CarouselShelf.kt
```

```
package com.example.holodex.ui.composables

import androidx.compose.foundation.background
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.PaddingValues
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.width
import androidx.compose.foundation.lazy.LazyRow
import androidx.compose.foundation.lazy.items
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.ErrorOutline
import androidx.compose.material3.Icon
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
import com.example.holodex.viewmodel.state.UiState

@Composable
fun <T> CarouselShelf(
    title: String,
    uiState: UiState<List<T>>,
    itemContent: @Composable (T) -> Unit,
    modifier: Modifier = Modifier,
    actionContent: (@Composable () -> Unit)? = null
) {
    Column(modifier = modifier) {
        Row(
            modifier = Modifier
                .fillMaxWidth()
                .padding(horizontal = 16.dp),
            verticalAlignment = Alignment.CenterVertically,
            horizontalArrangement = Arrangement.SpaceBetween
        ) {
            Text(text = title, style = MaterialTheme.typography.titleLarge)
            actionContent?.invoke()
        }

        when (uiState) {
            is UiState.Loading -> {
                LazyRow(
                    modifier = Modifier.fillMaxWidth(),
                    contentPadding = PaddingValues(horizontal = 16.dp),
                    horizontalArrangement = Arrangement.spacedBy(12.dp)
                ) {
                    items(5) {
                        Box(modifier = Modifier.width(140.dp).height(180.dp).background(Materi
```

```
                    }
                }
            }
            is UiState.Success -> {
                if (uiState.data.isEmpty()) {
                    Text(
                        text = "No content available.",
                        modifier = Modifier.padding(16.dp),
                        style = MaterialTheme.typography.bodyMedium,
                        color = MaterialTheme.colorScheme.onSurfaceVariant
                    )
                } else {
                    LazyRow(
                        modifier = Modifier.fillMaxWidth(),
                        contentPadding = PaddingValues(horizontal = 16.dp),
                        horizontalArrangement = Arrangement.spacedBy(12.dp)
                    ) {
                        items(uiState.data) { item ->
                            itemContent(item)
                        }
                    }
                }
            }
            is UiState.Error -> {
                Row(
                    modifier = Modifier.padding(16.dp),
                    verticalAlignment = Alignment.CenterVertically
                ) {
                    Icon(Icons.Default.ErrorOutline, contentDescription = "Error", tint = Mate
                    Spacer(Modifier.width(8.dp))
                    Text(
                        text = uiState.message,
                        color = MaterialTheme.colorScheme.error,
                        style = MaterialTheme.typography.bodyMedium
                    )
                }
            }
        }
    }
}

// File: java\com\example\holodex\ui\composables\ChannelCard.kt
package com.example.holodex.ui.composables

import androidx.compose.foundation.clickable
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.layout.width
import androidx.compose.foundation.shape.CircleShape
import androidx.compose.material3.Card
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
```

```
import androidx.compose.runtime.remember
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.layout.ContentScale
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.text.style.TextAlign
import androidx.compose.ui.text.style.TextOverflow
import androidx.compose.ui.unit.dp
import coil.compose.AsyncImage
import coil.request.ImageRequest
import com.example.holodex.R
import com.example.holodex.data.model.discovery.DiscoveryChannel
import com.example.holodex.util.ArtworkResolver

@Composable
fun ChannelCard(
    channel: DiscoveryChannel,
    onChannelClicked: (String) -> Unit,
    modifier: Modifier = Modifier
) {
    // --- START OF IMPLEMENTATION ---
    val artworkUrl = remember(channel.id, channel.photoUrl) {
        // Prioritize the photoUrl if it exists, otherwise construct it from the ID.
        channel.photoUrl?.takeIf { it.isNotBlank() }
            ?: ArtworkResolver.getChannelPhotoUrl(channel.id)
    }
    // --- END OF IMPLEMENTATION ---

    Card(
        modifier = modifier
            .width(140.dp)
            .clickable { onChannelClicked(channel.id) }
    ) {
        Column(
            modifier = Modifier.padding(12.dp),
            horizontalAlignment = Alignment.CenterHorizontally,
            verticalArrangement = Arrangement.spacedBy(8.dp)
        ) {
            AsyncImage(
                // --- MODIFICATION: Use the new artworkUrl variable ---
                model = ImageRequest.Builder(LocalContext.current)
                    .data(artworkUrl)
                    .placeholder(R.drawable.ic_placeholder_image)
                    .error(R.drawable.ic_error_image)
                    .crossfade(true).build(),
                contentDescription = "Avatar for ${channel.name}",
                modifier = Modifier.size(96.dp).clip(CircleShape),
                contentScale = ContentScale.Crop
            )
            Text(
                text = channel.englishName ?: channel.name,
                style = MaterialTheme.typography.titleSmall,
                textAlign = TextAlign.Center,
                maxLines = 2,
```

```
                overflow = TextOverflow.Ellipsis
            )
        }
    }
}

// File: java\com\example\holodex\ui\composables\CustomPagedUnifiedList.kt
@file:kotlin.OptIn(ExperimentalMaterial3Api::class)

package com.example.holodex.ui.composables

import androidx.annotation.OptIn
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.PaddingValues
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.LazyListState
import androidx.compose.foundation.lazy.items
import androidx.compose.material3.CircularProgressIndicator
import androidx.compose.material3.ExperimentalMaterial3Api
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Text
import androidx.compose.material3.pulltorefresh.PullToRefreshBox
import androidx.compose.runtime.Composable
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.runtime.derivedStateOf
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.text.style.TextAlign
import androidx.compose.ui.unit.dp
import androidx.media3.common.util.UnstableApi
import androidx.navigation.NavController
import com.example.holodex.R
import com.example.holodex.viewmodel.FavoritesViewModel
import com.example.holodex.viewmodel.PlaylistManagementViewModel
import com.example.holodex.viewmodel.UnifiedDisplayItem
import com.example.holodex.viewmodel.VideoListViewModel
import timber.log.Timber

@OptIn(UnstableApi::class, ExperimentalMaterial3Api::class) // Fixed: Removed duplicate annota
@Composable
fun CustomPagedUnifiedList(
    listKeyPrefix: String,
    items: List<UnifiedDisplayItem>,
    listState: LazyListState,
    onItemClicked: (UnifiedDisplayItem) -> Unit,
    videoListViewModel: VideoListViewModel,
```

```
    favoritesViewModel: FavoritesViewModel,
    playlistManagementViewModel: PlaylistManagementViewModel,
    navController: NavController,
    isLoadingMore: Boolean,
    endOfList: Boolean,
    isRefreshing: Boolean,
    onRefresh: () -> Unit,
    onLoadMore: () -> Unit,
    modifier: Modifier = Modifier,
    contentPadding: PaddingValues = PaddingValues(bottom = 80.dp),
    header: (@Composable () -> Unit)? = null,
) {
    // Track initial load state to fix scroll-to-end bug
    var hasPerformedInitialScroll by remember { mutableStateOf(false) }

    // Fix for scroll-to-end bug: Ensure we start at top after initial load
    LaunchedEffect(items.size) {
        if (items.isNotEmpty() && !hasPerformedInitialScroll) {
            // Only scroll to top if we're not already there (avoids unnecessary animation)
            if (listState.firstVisibleItemIndex != 0 || listState.firstVisibleItemScrollOffset
                listState.scrollToItem(0)
            }
            hasPerformedInitialScroll = true
        } else if (items.isEmpty()) {
            // Reset flag if list becomes empty (e.g., after refresh)
            hasPerformedInitialScroll = false
        }
    }

    // Improved load-more logic with better performance and stability
    val shouldLoadMore by remember {
        derivedStateOf {
            // Early exit conditions for better performance
            if (isLoadingMore || endOfList || items.isEmpty()) {
                false
            } else {
                val layoutInfo = listState.layoutInfo
                val visibleItemsInfo = layoutInfo.visibleItemsInfo

                // More robust check
                if (visibleItemsInfo.isEmpty()) {
                    false
                } else {
                    val lastVisibleItem = visibleItemsInfo.last()
                    val threshold = 3
                    val totalItems = layoutInfo.totalItemsCount

                    // Account for header in total count if present
                    val adjustedTotalItems = if (header != null) totalItems - 1 else totalItem

                    lastVisibleItem.index >= adjustedTotalItems - 1 - threshold
                }
            }
        }
    }
```

```
// More efficient LaunchedEffect that only triggers when actually needed
LaunchedEffect(shouldLoadMore, isLoadingMore, endOfList) {
    if (shouldLoadMore && !isLoadingMore && !endOfList) {
        Timber.i("CustomPagedUnifiedList ($listKeyPrefix): >>> LOAD MORE UI TRIGGERED <<<"
        onLoadMore()
    }
}

PullToRefreshBox(
    isRefreshing = isRefreshing,
    onRefresh = {
        // Reset initial scroll flag on refresh
        hasPerformedInitialScroll = false
        onRefresh()
    },
    modifier = modifier
) {
    LazyColumn(
        state = listState,
        modifier = Modifier.fillMaxSize(), // Removed redundant modifier parameter
        contentPadding = contentPadding
    ) {
        header?.let {
            item(key = "${listKeyPrefix}_header") { it() }
        }

        items(
            items = items,
            key = { item -> item.stableId }
        ) { item ->
            UnifiedListItem(
                item = item,
                onItemClicked = { onItemClicked(item) }, // Fixed formatting
                videoListViewModel = videoListViewModel,
                favoritesViewModel = favoritesViewModel,
                playlistManagementViewModel = playlistManagementViewModel,
                navController = navController,
            )
        }

        item(key = "${listKeyPrefix}_footer") {
            if (isLoadingMore) {
                Box(
                    modifier = Modifier
                        .fillMaxWidth()
                        .padding(vertical = 16.dp),
                    contentAlignment = Alignment.Center
                ) {
                    CircularProgressIndicator(modifier = Modifier.size(36.dp))
                }
            } else if (endOfList && items.isNotEmpty()) {
                Text(
                    text = stringResource(R.string.message_youve_reached_the_end),
                    modifier = Modifier
```

```
                            .fillMaxWidth()
                            .padding(16.dp), // Fixed chaining
                        textAlign = TextAlign.Center,
                        style = MaterialTheme.typography.bodySmall,
                        color = MaterialTheme.colorScheme.onSurfaceVariant
                    )
                }
            }
        }
    }
}

// File: java\com\example\holodex\ui\composables\FullPlayerScreen.kt
// File: java/com/example/holodex/ui/composables/FullPlayerScreen.kt
package com.example.holodex.ui.composables

import android.content.Context
import android.media.AudioManager
import android.widget.Toast
import androidx.compose.animation.AnimatedContent
import androidx.compose.animation.AnimatedVisibility
import androidx.compose.animation.animateColorAsState
import androidx.compose.animation.core.Animatable
import androidx.compose.animation.core.EaseOutCubic
import androidx.compose.animation.core.animateDpAsState
import androidx.compose.animation.core.tween
import androidx.compose.animation.fadeIn
import androidx.compose.animation.fadeOut
import androidx.compose.animation.scaleIn
import androidx.compose.animation.scaleOut
import androidx.compose.animation.togetherWith
import androidx.compose.foundation.BorderStroke
import androidx.compose.foundation.ExperimentalFoundationApi
import androidx.compose.foundation.background
import androidx.compose.foundation.combinedClickable
import androidx.compose.foundation.gestures.detectDragGestures
import androidx.compose.foundation.interaction.MutableInteractionSource
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.PaddingValues
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.aspectRatio
import androidx.compose.foundation.layout.fillMaxHeight
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.navigationBarsPadding
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.layout.systemBarsPadding
import androidx.compose.foundation.layout.width
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.itemsIndexed
```

```
import androidx.compose.foundation.lazy.rememberLazyListState
import androidx.compose.foundation.rememberScrollState
import androidx.compose.foundation.shape.CircleShape
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.foundation.verticalScroll
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.automirrored.filled.ArrowBack
import androidx.compose.material.icons.automirrored.filled.PlaylistAdd
import androidx.compose.material.icons.automirrored.filled.PlaylistPlay
import androidx.compose.material.icons.automirrored.filled.VolumeDown
import androidx.compose.material.icons.automirrored.filled.VolumeMute
import androidx.compose.material.icons.automirrored.filled.VolumeUp
import androidx.compose.material.icons.automirrored.outlined.QueueMusic
import androidx.compose.material.icons.filled.Delete
import androidx.compose.material.icons.filled.DragHandle
import androidx.compose.material.icons.filled.Equalizer
import androidx.compose.material.icons.filled.Favorite
import androidx.compose.material.icons.filled.FavoriteBorder
import androidx.compose.material.icons.filled.MoreVert
import androidx.compose.material.icons.filled.MusicNote
import androidx.compose.material.icons.filled.PersonSearch
import androidx.compose.material.icons.filled.PlayArrow
import androidx.compose.material.icons.filled.PlaylistRemove
import androidx.compose.material.icons.filled.TextFields
import androidx.compose.material3.CircularProgressIndicator
import androidx.compose.material3.DropdownMenu
import androidx.compose.material3.DropdownMenuItem
import androidx.compose.material3.ExperimentalMaterial3Api
import androidx.compose.material3.FilledTonalIconButton
import androidx.compose.material3.HorizontalDivider
import androidx.compose.material3.Icon
import androidx.compose.material3.IconButton
import androidx.compose.material3.IconButtonDefaults
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.ModalBottomSheet
import androidx.compose.material3.Slider
import androidx.compose.material3.SliderDefaults
import androidx.compose.material3.Surface
import androidx.compose.material3.Text
import androidx.compose.material3.rememberModalBottomSheetState
import androidx.compose.runtime.Composable
import androidx.compose.runtime.Immutable
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableFloatStateOf
import androidx.compose.runtime.mutableIntStateOf
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.rememberCoroutineScope
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.graphics.graphicsLayer
```

```
import androidx.compose.ui.hapticfeedback.HapticFeedbackType
import androidx.compose.ui.input.pointer.pointerInput
import androidx.compose.ui.layout.ContentScale
import androidx.compose.ui.layout.onSizeChanged
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.platform.LocalDensity
import androidx.compose.ui.platform.LocalHapticFeedback
import androidx.compose.ui.res.pluralStringResource
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.text.style.TextAlign
import androidx.compose.ui.text.style.TextOverflow
import androidx.compose.ui.unit.IntSize
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import androidx.hilt.navigation.compose.hiltViewModel
import androidx.lifecycle.compose.collectAsStateWithLifecycle
import androidx.media3.common.Player
import androidx.media3.common.util.UnstableApi
import androidx.navigation.NavHostController
import coil.compose.AsyncImage
import coil.request.ImageRequest
import com.example.holodex.R
import com.example.holodex.playback.domain.model.PlaybackItem
import com.example.holodex.playback.util.formatDurationSecondsToString
import com.example.holodex.ui.AppDestinations
import com.example.holodex.util.ThumbnailQuality
import com.example.holodex.util.generateArtworkUrlList
import com.example.holodex.viewmodel.FavoritesViewModel
import com.example.holodex.viewmodel.FullPlayerViewModel
import com.example.holodex.viewmodel.PlaybackViewModel
import com.example.holodex.viewmodel.rememberFullPlayerArtworkState
import com.example.holodex.viewmodel.rememberFullPlayerCurrentItemState
import com.example.holodex.viewmodel.rememberFullPlayerLoadingState
import com.example.holodex.viewmodel.rememberFullPlayerProgressState
import com.example.holodex.viewmodel.rememberFullPlayerQueueInfoState
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch
import org.orbitmvi.orbit.compose.collectAsState
import sh.calvin.reorderable.ReorderableItem
import sh.calvin.reorderable.rememberReorderableLazyListState
import kotlin.math.absoluteValue
import kotlin.math.roundToInt

private object FullPlayerDefaults {
    const val VOLUME_SWIPE_SENSITIVITY_FACTOR = 0.005f
    const val HORIZONTAL_SWIPE_THRESHOLD_PX = 50f
    val ArtworkShape = RoundedCornerShape(16.dp)
    const val ARTWORK_ANIMATION_DURATION_MS = 250
    const val VOLUME_SLIDER_AUTO_HIDE_DELAY_MS = 3000L
    const val QUEUE_SHEET_MAX_HEIGHT_FACTOR = 0.7f
}

@Immutable
data class FullPlayerActions(
```

```
    val onNavigateUp: () -> Unit,
    val onTogglePlayPause: () -> Unit,
    val onSeekTo: (Long) -> Unit,
    val onSkipToNext: () -> Unit,
    val onSkipToPrevious: () -> Unit,
    val onToggleRepeatMode: () -> Unit,
    val onToggleShuffleMode: () -> Unit,
    val onPlayQueueItemAtIndex: (Int) -> Unit,
    val onReorderQueueItem: (from: Int, to: Int) -> Unit,
    val onRemoveQueueItem: (index: Int) -> Unit,
    val onClearQueue: () -> Unit,
    val onToggleLike: (PlaybackItem) -> Unit,
    val onFindArtist: (channelId: String) -> Unit,
    val onOpenAudioSettings: (audioSessionId: Int) -> Unit,
    val onAddToPlaylist: (PlaybackItem) -> Unit,
)

@UnstableApi
@OptIn(ExperimentalMaterial3Api::class, ExperimentalFoundationApi::class)
@Composable
internal fun FullPlayerScreenContent(
    navController: NavHostController,
    player: Player?,
    actions: FullPlayerActions,
    modifier: Modifier = Modifier
) {
    val coroutineScope = rememberCoroutineScope()
    val context = LocalContext.current

    // ViewModels
    val fullPlayerViewModel: FullPlayerViewModel = hiltViewModel()
    val playbackViewModel: PlaybackViewModel = hiltViewModel()
    val favoritesViewModel: FavoritesViewModel = hiltViewModel()

    // State from ViewModels
    val isRadioMode by playbackViewModel.isRadioModeActive.collectAsStateWithLifecycle()
    val artworkUri by rememberFullPlayerArtworkState(playbackViewModel.uiState)
    val currentItem by rememberFullPlayerCurrentItemState(playbackViewModel.uiState)
    val isLoading by rememberFullPlayerLoadingState(playbackViewModel.uiState)
    val queueInfo by rememberFullPlayerQueueInfoState(playbackViewModel.uiState)
    val (queueItems, currentIndexInQueue, isQueueNotEmpty) = queueInfo
    val uiState by playbackViewModel.uiState.collectAsStateWithLifecycle()
    val repeatMode = uiState.repeatMode
    val shuffleMode = uiState.shuffleMode
    val progress by rememberFullPlayerProgressState(playbackViewModel.uiState)
    val favoritesState by favoritesViewModel.collectAsState()
    val dynamicTheme by fullPlayerViewModel.dynamicTheme.collectAsStateWithLifecycle()

    // Local state
    var showVolumeSlider by remember { mutableStateOf(false) }
    var volumeSliderValue by remember { mutableFloatStateOf(0.7f) }
    var showLyricsView by remember { mutableStateOf(false) }
    var showQueueSheet by remember { mutableStateOf(false) }

    // Animation state
```

```
val artworkScale = remember { Animatable(1f) }
val artworkAlpha = remember { Animatable(1f) }

// Animated colors based on dynamic theme
val animatedPrimaryColor by animateColorAsState(
    dynamicTheme.primary,
    label = "animated_primary_color",
    animationSpec = tween(1200)
)
val animatedOnPrimaryColor by animateColorAsState(
    dynamicTheme.onPrimary,
    label = "animated_on_primary_color",
    animationSpec = tween(500)
)

// Audio and haptic feedback
val audioManager = remember { context.getSystemService(Context.AUDIO_SERVICE) as AudioMana
val maxVolume = remember { audioManager.getStreamMaxVolume(AudioManager.STREAM_MUSIC) }
val haptic = LocalHapticFeedback.current

// Computed states
val isCurrentItemLiked = remember(currentItem, favoritesState.likedItemsMap) {
    currentItem?.let { pbItem ->
        val likeId = favoritesViewModel.getLikeIdForPlaybackItem(pbItem)
        favoritesState.likedItemsMap.containsKey(likeId)
    } == true
}

val queueSheetState = rememberModalBottomSheetState(skipPartiallyExpanded = true)

// Update theme when artwork changes
LaunchedEffect(artworkUri) {
    fullPlayerViewModel.updateThemeFromArtwork(artworkUri)
}

// Artwork transition animations
LaunchedEffect(currentItem?.id) {
    if (currentItem != null) {
        coroutineScope.launch {
            artworkAlpha.animateTo(0.5f, tween(150))
            artworkAlpha.animateTo(1f, tween(FullPlayerDefaults.ARTWORK_ANIMATION_DURATION
        }
        coroutineScope.launch {
            artworkScale.animateTo(0.95f, tween(150))
            artworkScale.animateTo(1f, tween(FullPlayerDefaults.ARTWORK_ANIMATION_DURATION
        }
    }
}

// Auto-hide volume slider
LaunchedEffect(showVolumeSlider) {
    if (showVolumeSlider) {
        delay(FullPlayerDefaults.VOLUME_SLIDER_AUTO_HIDE_DELAY_MS)
        showVolumeSlider = false
    }
```

```
}

Box(modifier = modifier.fillMaxSize()) {
    // Background with dynamic theme
    SimpleProcessedBackground(
        artworkUri = artworkUri,
        dynamicColor = dynamicTheme.primary
    )
    Surface(
        modifier = Modifier.fillMaxSize(),
        color = animatedPrimaryColor.copy(alpha = 0.45f)
    ) {}

    Column(
        modifier = Modifier
            .fillMaxSize()
            .systemBarsPadding()
    ) {
        PlayerTopBar(
            isLiked = isCurrentItemLiked,
            isShowingLyrics = showLyricsView,
            queueNotEmpty = isQueueNotEmpty,
            isItemLoaded = currentItem != null,
            iconTint = animatedOnPrimaryColor,
            onNavigateUp = actions.onNavigateUp,
            onLikeToggle = { currentItem?.let { actions.onToggleLike(it) } },
            onQueueClick = { coroutineScope.launch { showQueueSheet = true } },
            onToggleLyrics = { showLyricsView = !showLyricsView },
            actions = actions,
            currentItem = currentItem,
            navController = navController,
            playbackViewModel = playbackViewModel
        )

        Box(
            modifier = Modifier
                .weight(1f)
                .fillMaxWidth()
        ) {
            if (showLyricsView) {
                LyricsView(
                    currentItem = currentItem,
                    textColor = animatedOnPrimaryColor,
                    modifier = Modifier.fillMaxSize().padding(16.dp)
                )
            } else {
                PlayerContent(
                    currentItem = currentItem,
                    trackInfoColor = animatedOnPrimaryColor,
                    isLoading = isLoading,
                    artworkScale = artworkScale.value,
                    artworkAlpha = artworkAlpha.value,
                    onHorizontalSwipe = { dragAmount ->
                        if (dragAmount < -FullPlayerDefaults.HORIZONTAL_SWIPE_THRESHOLD_PX
                            haptic.performHapticFeedback(HapticFeedbackType.TextHandleMove
```

```
                        actions.onSkipToNext()
                    } else if (dragAmount > FullPlayerDefaults.HORIZONTAL_SWIPE_THRESH
                        haptic.performHapticFeedback(HapticFeedbackType.TextHandleMove
                        actions.onSkipToPrevious()
                    }
                },
                onVerticalSwipe = { deltaY ->
                    val currentVolume = audioManager.getStreamVolume(AudioManager.STRE
                    val newVolume = (currentVolume - deltaY * FullPlayerDefaults.VOLUM
                        .roundToInt().coerceIn(0, maxVolume)
                    if (newVolume != currentVolume) {
                        audioManager.setStreamVolume(AudioManager.STREAM_MUSIC, newVol
                        volumeSliderValue = newVolume.toFloat() / maxVolume
                        if (!showVolumeSlider) showVolumeSlider = true
                        haptic.performHapticFeedback(HapticFeedbackType.TextHandleMove
                    }
                },
                onDoubleTap = {
                    haptic.performHapticFeedback(HapticFeedbackType.LongPress)
                    actions.onTogglePlayPause()
                },
                modifier = Modifier.fillMaxSize()
            )
        }
    }

    player?.let { validPlayer ->
        AnimatedVisibility(visible = showVolumeSlider && !showLyricsView) {
            VolumeSlider(
                value = volumeSliderValue,
                onValueChange = {
                    volumeSliderValue = it
                    val newVolumeInt = (it * maxVolume).roundToInt()
                    audioManager.setStreamVolume(AudioManager.STREAM_MUSIC, newVolumeI
                },
                thumbColor = animatedPrimaryColor,
                activeTrackColor = animatedOnPrimaryColor
            )
        }

        Media3PlayerControls(
            player = validPlayer,
            progress = progress,
            shuffleMode = shuffleMode,
            repeatMode = repeatMode,
            onSeek = { positionSec -> actions.onSeekTo(positionSec) },
            onScrubbingChange = { isScrubbing -> playbackViewModel.setScrubbing(isScru
            primaryColor = animatedPrimaryColor,
            onPrimaryColor = animatedOnPrimaryColor,
            onPlayPause = actions.onTogglePlayPause,
            onSkipPrevious = actions.onSkipToPrevious,
            onSkipNext = actions.onSkipToNext,
            onToggleShuffle = actions.onToggleShuffleMode,
            isRadioMode = isRadioMode,
            onToggleRepeat = actions.onToggleRepeatMode
```

```
                )
            }
        }

        if (showQueueSheet) {
            ModalBottomSheet(
                onDismissRequest = { showQueueSheet = false },
                sheetState = queueSheetState
            ) {
                Box(modifier = Modifier.fillMaxHeight(FullPlayerDefaults.QUEUE_SHEET_MAX_HEIGH
                    QueueSheetContent(
                        queueItems = queueItems,
                        currentIndex = currentIndexInQueue,
                        isRadioMode = isRadioMode,
                        onPlayQueueItem = actions.onPlayQueueItemAtIndex,
                        onClearQueue = {
                            actions.onClearQueue()
                            coroutineScope.launch { queueSheetState.hide(); showQueueSheet = f
                        },
                        onMoveItem = actions.onReorderQueueItem,
                        onRemoveItem = actions.onRemoveQueueItem
                    )
                }
            }
        }
    }
}

@Composable
private fun PlayerTopBar(
    isLiked: Boolean,
    isShowingLyrics: Boolean,
    queueNotEmpty: Boolean,
    isItemLoaded: Boolean,
    onNavigateUp: () -> Unit,
    onLikeToggle: () -> Unit,
    onQueueClick: () -> Unit,
    onToggleLyrics: () -> Unit,
    iconTint: Color,
    actions: FullPlayerActions,
    currentItem: PlaybackItem?,
    navController: NavHostController,
    playbackViewModel: PlaybackViewModel
) {
    var showMoreOptionsDropdown by remember { mutableStateOf(false) }

    Row(
        modifier = Modifier
            .fillMaxWidth()
            .padding(horizontal = 4.dp, vertical = 8.dp),
        verticalAlignment = Alignment.CenterVertically,
    ) {
        IconButton(onClick = onNavigateUp) {
            Icon(Icons.AutoMirrored.Filled.ArrowBack, stringResource(R.string.content_desc_nav
        }
```

```
        Spacer(Modifier.weight(1f))

        IconButton(onClick = onToggleLyrics) {
            Icon(
                imageVector = if (isShowingLyrics) Icons.Filled.MusicNote else Icons.Filled.Te
                contentDescription = stringResource(if (isShowingLyrics) R.string.action_hide_
                tint = iconTint
            )
        }

        IconButton(onClick = onLikeToggle, enabled = isItemLoaded) {
            Icon(
                imageVector = if (isLiked) Icons.Filled.Favorite else Icons.Filled.FavoriteBor
                contentDescription = stringResource(if (isLiked) R.string.content_desc_unlike_
                tint = iconTint
            )
        }

        IconButton(onClick = onQueueClick, enabled = queueNotEmpty) {
            Icon(Icons.AutoMirrored.Filled.PlaylistPlay, stringResource(R.string.content_desc_
        }

        Box {
            IconButton(onClick = { showMoreOptionsDropdown = true }, enabled = isItemLoaded) {
                Icon(Icons.Filled.MoreVert, stringResource(R.string.content_desc_more_options)
            }
            PlayerOverflowMenu(
                expanded = showMoreOptionsDropdown,
                onDismissRequest = { showMoreOptionsDropdown = false },
                currentItem = currentItem,
                actions = actions,
                navController = navController,
                playbackViewModel = playbackViewModel
            )
        }
    }
}

@Composable
private fun PlayerOverflowMenu(
    expanded: Boolean,
    onDismissRequest: () -> Unit,
    currentItem: PlaybackItem?,
    actions: FullPlayerActions,
    navController: NavHostController,
    playbackViewModel: PlaybackViewModel
) {
    val context = LocalContext.current
    DropdownMenu(
        expanded = expanded,
        onDismissRequest = onDismissRequest
    ) {
        DropdownMenuItem(
            text = { Text(stringResource(R.string.action_add_to_playlist_menu)) },
            onClick = {
```

```
                currentItem?.let { actions.onAddToPlaylist(it) }
                onDismissRequest()
            },
            leadingIcon = { Icon(Icons.AutoMirrored.Filled.PlaylistAdd, null) },
            enabled = currentItem != null
        )
        val artistChannelId = currentItem?.channelId
        DropdownMenuItem(
            text = { Text(stringResource(R.string.action_view_artist)) },
            onClick = {
                if (!artistChannelId.isNullOrBlank()) {
                    actions.onFindArtist(artistChannelId)
                    navController.navigate(AppDestinations.HOME_ROUTE) {
                        popUpTo(navController.graph.startDestinationRoute ?: AppDestinations.H
                        launchSingleTop = true; restoreState = true
                    }
                }
                onDismissRequest()
            },
            leadingIcon = { Icon(Icons.Filled.PersonSearch, null) },
            enabled = !artistChannelId.isNullOrBlank()
        )
        DropdownMenuItem(
            text = { Text(stringResource(R.string.action_audio_settings)) },
            onClick = {
                playbackViewModel.getAudioSessionId()?.let {
                    if (it != 0) actions.onOpenAudioSettings(it)
                    else Toast.makeText(context, R.string.error_no_audio_session, Toast.LENGTH
                }
                onDismissRequest()
            },
            leadingIcon = { Icon(Icons.Filled.Equalizer, null) }
        )
    }
}

@OptIn(ExperimentalFoundationApi::class)
@Composable
private fun PlayerContent(
    modifier: Modifier = Modifier,
    currentItem: PlaybackItem?,
    trackInfoColor: Color,
    isLoading: Boolean,
    artworkScale: Float,
    artworkAlpha: Float,
    onHorizontalSwipe: (dragAmount: Float) -> Unit,
    onVerticalSwipe: (deltaY: Float) -> Unit,
    onDoubleTap: () -> Unit
) {
    val context = LocalContext.current

    // Use onSizeChanged to determine artwork size based on parent container
    var parentSize by remember { mutableStateOf(IntSize.Zero) }
    val artworkSize = with(LocalDensity.current) { (parentSize.height * 0.4f).toDp() }
```

```
val artworkUrls = remember(currentItem) {
    generateArtworkUrlList(currentItem, ThumbnailQuality.MAX)
}
var currentUrlIndex by remember(artworkUrls) { mutableIntStateOf(0) }

Column(
    modifier = modifier
        .fillMaxWidth()
        .padding(horizontal = 24.dp, vertical = 16.dp)
        .onSizeChanged { parentSize = it },
    horizontalAlignment = Alignment.CenterHorizontally,
    verticalArrangement = Arrangement.SpaceAround
) {
    Spacer(modifier = Modifier.height(artworkSize * 0.1f))
    Box(
        modifier = Modifier
            .size(artworkSize)
            .aspectRatio(1f)
            .clip(FullPlayerDefaults.ArtworkShape)
            .background(MaterialTheme.colorScheme.surfaceVariant.copy(alpha = 0.3f))
            .graphicsLayer { scaleX = artworkScale; scaleY = artworkScale; alpha = artwork
            .combinedClickable(
                interactionSource = remember { MutableInteractionSource() },
                indication = null,
                onDoubleClick = onDoubleTap,
                onClick = {}
            )
            .pointerInput(Unit) {
                detectDragGestures { change, dragAmount ->
                    change.consume()
                    if (dragAmount.y.absoluteValue > dragAmount.x.absoluteValue * 1.5) {
                        onVerticalSwipe(dragAmount.y)
                    } else {
                        onHorizontalSwipe(dragAmount.x)
                    }
                }
            },
        contentAlignment = Alignment.Center
    ) {
        if (isLoading && currentItem == null) {
            CircularProgressIndicator()
        } else {
            AsyncImage(
                model = ImageRequest.Builder(context)
                    .data(artworkUrls.getOrNull(currentUrlIndex))
                    .placeholder(R.drawable.ic_default_album_art_placeholder)
                    .error(R.drawable.ic_error_image)
                    .crossfade(true).build(),
                contentDescription = stringResource(R.string.content_desc_album_art),
                contentScale = ContentScale.Crop,
                modifier = Modifier.fillMaxSize(),
                onError = { if (currentUrlIndex < artworkUrls.lastIndex) { currentUrlIndex
            )
        }
    }
```

```
        Spacer(modifier = Modifier.height(artworkSize * 0.15f))
        TrackInfo(currentItem, trackInfoColor)
        Spacer(modifier = Modifier.weight(1f))
    }
}

@Composable
private fun VolumeSlider(
    value: Float,
    onValueChange: (Float) -> Unit,
    thumbColor: Color,
    activeTrackColor: Color,
    modifier: Modifier = Modifier
) {
    Row(
        modifier = modifier
            .fillMaxWidth()
            .padding(horizontal = 24.dp, vertical = 8.dp),
        verticalAlignment = Alignment.CenterVertically
    ) {
        val icon = when {
            value < 0.01f -> Icons.AutoMirrored.Filled.VolumeMute
            value < 0.5f -> Icons.AutoMirrored.Filled.VolumeDown
            else -> Icons.AutoMirrored.Filled.VolumeUp
        }
        Icon(
            imageVector = icon,
            contentDescription = stringResource(R.string.content_desc_volume),
            modifier = Modifier.size(24.dp),
            tint = activeTrackColor
        )
        Slider(
            value = value,
            onValueChange = onValueChange,
            modifier = Modifier.weight(1f).padding(horizontal = 8.dp),
            valueRange = 0f..1f,
            colors = SliderDefaults.colors(
                thumbColor = thumbColor,
                activeTrackColor = activeTrackColor,
                inactiveTrackColor = activeTrackColor.copy(alpha = 0.3f)
            )
        )
    }
}

@Composable
private fun TrackInfo(currentItem: PlaybackItem?, textColor: Color) {
    AnimatedContent(
        targetState = currentItem?.id ?: "loading",
        transitionSpec = { fadeIn(tween(220, 90)) togetherWith fadeOut(tween(90)) },
        label = "trackInfoAnimation"
    ) { targetId ->
        Column(
            horizontalAlignment = Alignment.CenterHorizontally,
            modifier = Modifier.fillMaxWidth().padding(horizontal = 16.dp)
```

```
        ) {
            Text(
                text = if (targetId == "loading" || currentItem == null) stringResource(R.stri
                color = textColor,
                style = MaterialTheme.typography.headlineSmall.copy(fontSize = 22.sp),
                fontWeight = FontWeight.SemiBold,
                textAlign = TextAlign.Center,
                maxLines = 2,
                overflow = TextOverflow.Ellipsis
            )
            Spacer(Modifier.height(4.dp))
            Text(
                text = if (targetId == "loading" || currentItem == null) "" else currentItem.a
                color = textColor.copy(alpha = 0.8f),
                style = MaterialTheme.typography.titleMedium.copy(fontSize = 17.sp, lineHeight
                textAlign = TextAlign.Center,
                maxLines = 2,
                overflow = TextOverflow.Ellipsis
            )
        }
    }
}

@Composable
private fun LyricsView(currentItem: PlaybackItem?, modifier: Modifier = Modifier, textColor: C
    Box(modifier = modifier.fillMaxSize(), contentAlignment = Alignment.Center) {
        if (currentItem?.description.isNullOrBlank()) {
            Text(
                text = stringResource(R.string.lyrics_not_available),
                style = MaterialTheme.typography.bodyLarge,
                textAlign = TextAlign.Center,
                modifier = Modifier.padding(16.dp),
                color = MaterialTheme.colorScheme.onSurfaceVariant
            )
        } else {
            Text(
                text = currentItem.description,
                style = MaterialTheme.typography.bodyLarge,
                color = textColor,
                modifier = Modifier.fillMaxSize().verticalScroll(rememberScrollState()).paddin
            )
        }
    }
}

@OptIn(ExperimentalMaterial3Api::class)
@Composable
private fun QueueSheetContent(
    queueItems: List<PlaybackItem>,
    currentIndex: Int,
    isRadioMode: Boolean,
    onPlayQueueItem: (Int) -> Unit,
    onClearQueue: () -> Unit,
    onMoveItem: (from: Int, to: Int) -> Unit,
    onRemoveItem: (index: Int) -> Unit
```

```
) {
    val listState = rememberLazyListState()
    val reorderableState = rememberReorderableLazyListState(
        lazyListState = listState,
        onMove = { from, to -> onMoveItem(from.index, to.index) }
    )

    LaunchedEffect(currentIndex) {
        if (currentIndex in 0..queueItems.lastIndex) {
            listState.animateScrollToItem(currentIndex)
        }
    }

    Column(modifier = Modifier.fillMaxSize().navigationBarsPadding()) {
        Surface(
            modifier = Modifier.fillMaxWidth(),
            color = MaterialTheme.colorScheme.surface,
            shadowElevation = 4.dp
        ) {
            Column {
                Box(
                    modifier = Modifier.fillMaxWidth().padding(vertical = 8.dp),
                    contentAlignment = Alignment.Center
                ) {
                    Surface(
                        modifier = Modifier.width(32.dp).height(4.dp),
                        shape = RoundedCornerShape(2.dp),
                        color = MaterialTheme.colorScheme.onSurfaceVariant.copy(alpha = 0.4f)
                    ) {}
                }
                Row(
                    modifier = Modifier.fillMaxWidth().padding(horizontal = 16.dp, vertical =
                    verticalAlignment = Alignment.CenterVertically,
                    horizontalArrangement = Arrangement.SpaceBetween
                ) {
                    Column {
                        Text(
                            text = stringResource(R.string.up_next_queue_title),
                            style = MaterialTheme.typography.titleLarge,
                            color = MaterialTheme.colorScheme.onSurface
                        )
                        if (queueItems.isNotEmpty()) {
                            Text(
                                text = pluralStringResource(
                                    R.plurals.queue_items_count,
                                    queueItems.size,
                                    queueItems.size
                                ),
                                style = MaterialTheme.typography.bodyMedium,
                                color = MaterialTheme.colorScheme.onSurfaceVariant
                            )
                        }
                    }
                    AnimatedVisibility(
                        visible = queueItems.isNotEmpty(),
```

```
                enter = fadeIn() + scaleIn(),
                exit = fadeOut() + scaleOut()
            ) {
                FilledTonalIconButton(
                    onClick = onClearQueue,
                    colors = IconButtonDefaults.filledTonalIconButtonColors(
                        containerColor = MaterialTheme.colorScheme.errorContainer,
                        contentColor = MaterialTheme.colorScheme.onErrorContainer
                    )
                ) {
                    Icon(
                        Icons.Filled.PlaylistRemove,
                        contentDescription = stringResource(R.string.action_clear_queu
                    )
                }
            }
        }
        HorizontalDivider(color = MaterialTheme.colorScheme.outlineVariant)
    }
}

AnimatedContent(
    targetState = queueItems.isEmpty(),
    transitionSpec = {
        fadeIn(animationSpec = tween(300)) togetherWith fadeOut(
            animationSpec = tween(300)
        )
    },
    label = "queue_content"
) { isEmpty ->
    if (isEmpty) {
        Box(
            modifier = Modifier.fillMaxSize().padding(32.dp),
            contentAlignment = Alignment.Center
        ) {
            Column(
                horizontalAlignment = Alignment.CenterHorizontally,
                verticalArrangement = Arrangement.spacedBy(16.dp)
            ) {
                Icon(
                    Icons.AutoMirrored.Outlined.QueueMusic,
                    null,
                    modifier = Modifier.size(64.dp),
                    tint = MaterialTheme.colorScheme.onSurfaceVariant.copy(alpha = 0.6
                )
                Text(
                    text = stringResource(R.string.empty_queue),
                    style = MaterialTheme.typography.titleMedium,
                    color = MaterialTheme.colorScheme.onSurfaceVariant,
                    textAlign = TextAlign.Center
                )
                Text(
                    text = stringResource(R.string.empty_queue_description),
                    style = MaterialTheme.typography.bodyMedium,
                    color = MaterialTheme.colorScheme.onSurfaceVariant.copy(alpha = 0.
```

```
                            textAlign = TextAlign.Center
                        )
                    }
                }
            } else {
                LazyColumn(
                    state = listState,
                    modifier = Modifier.fillMaxSize(),
                    contentPadding = PaddingValues(
                        start = 8.dp,
                        end = 8.dp,
                        top = 8.dp,
                        bottom = 16.dp
                    ),
                    verticalArrangement = Arrangement.spacedBy(4.dp)
                ) {
                    itemsIndexed(
                        items = queueItems,
                        key = { index, item -> "${item.id}_$index" }
                    ) { index, itemData ->
                        ReorderableItem(
                            state = reorderableState,
                            key = "${itemData.id}_$index",
                            enabled = !isRadioMode
                        ) { isDragging ->
                            QueueItemRow(
                                item = itemData,
                                index = index + 1,
                                isCurrentlyPlaying = index == currentIndex,
                                isDragging = isDragging,
                                isRadioMode = isRadioMode,
                                onItemClick = { onPlayQueueItem(index) },
                                onRemoveClick = { onRemoveItem(index) },
                                modifier = Modifier.longPressDraggableHandle()
                            )
                        }
                    }
                }
            }
        }
    }
}

@Composable
private fun QueueItemRow(
    item: PlaybackItem,
    index: Int,
    isCurrentlyPlaying: Boolean,
    isDragging: Boolean,
    isRadioMode: Boolean,
    onItemClick: () -> Unit,
    onRemoveClick: () -> Unit,
    modifier: Modifier = Modifier
) {
    val elevation by animateDpAsState(
```

```
    targetValue = if (isDragging) 8.dp else 0.dp,
    animationSpec = tween(200),
    label = "drag_elevation"
)
val containerColor by animateColorAsState(
    targetValue = when {
        isCurrentlyPlaying -> MaterialTheme.colorScheme.primaryContainer
        isDragging -> MaterialTheme.colorScheme.surfaceVariant
        else -> MaterialTheme.colorScheme.surface
    },
    animationSpec = tween(200),
    label = "container_color"
)

Surface(
    onClick = onItemClick,
    modifier = modifier.fillMaxWidth(),
    shape = RoundedCornerShape(12.dp),
    color = containerColor,
    shadowElevation = elevation,
    border = if (isCurrentlyPlaying) BorderStroke(2.dp, MaterialTheme.colorScheme.primary.
) {
    Row(
        modifier = Modifier.fillMaxWidth().padding(horizontal = 12.dp, vertical = 8.dp),
        verticalAlignment = Alignment.CenterVertically,
        horizontalArrangement = Arrangement.spacedBy(12.dp)
    ) {
        // Index/Play indicator
        Surface(
            shape = CircleShape,
            color = if (isCurrentlyPlaying) MaterialTheme.colorScheme.primary else Materia
            modifier = Modifier.size(32.dp)
        ) {
            Box(contentAlignment = Alignment.Center) {
                if (isCurrentlyPlaying) {
                    Icon(
                        Icons.Filled.PlayArrow,
                        null,
                        tint = MaterialTheme.colorScheme.onPrimary,
                        modifier = Modifier.size(16.dp)
                    )
                } else {
                    Text(
                        text = index.toString(),
                        style = MaterialTheme.typography.labelMedium,
                        color = MaterialTheme.colorScheme.onSurfaceVariant
                    )
                }
            }
        }

        // Artwork
        AsyncImage(
            model = ImageRequest.Builder(LocalContext.current)
                .data(item.artworkUri)
```

```
            .placeholder(R.drawable.ic_default_album_art_placeholder)
            .error(R.drawable.ic_error_image)
            .crossfade(true)
            .build(),
        contentDescription = null,
        modifier = Modifier.size(48.dp).clip(RoundedCornerShape(8.dp)),
        contentScale = ContentScale.Crop
    )

    // Title and artist
    Column(
        modifier = Modifier.weight(1f),
        verticalArrangement = Arrangement.spacedBy(2.dp)
    ) {
        Text(
            text = item.title,
            style = MaterialTheme.typography.bodyLarge,
            color = if (isCurrentlyPlaying) MaterialTheme.colorScheme.onPrimaryContain
            maxLines = 1,
            overflow = TextOverflow.Ellipsis,
            fontWeight = if (isCurrentlyPlaying) FontWeight.Medium else FontWeight.Nor
        )
        Text(
            text = item.artistText,
            style = MaterialTheme.typography.bodyMedium,
            color = if (isCurrentlyPlaying) MaterialTheme.colorScheme.onPrimaryContain
            maxLines = 1,
            overflow = TextOverflow.Ellipsis
        )
    }

    // Duration
    Text(
        text = formatDurationSecondsToString(item.durationSec),
        style = MaterialTheme.typography.bodySmall,
        color = MaterialTheme.colorScheme.onSurfaceVariant
    )

    // Remove button
    IconButton(
        onClick = onRemoveClick,
        modifier = Modifier.size(40.dp)
    ) {
        Icon(
            Icons.Default.Delete,
            contentDescription = stringResource(R.string.action_remove_from_queue),
            tint = MaterialTheme.colorScheme.onSurfaceVariant.copy(alpha = 0.7f),
            modifier = Modifier.size(20.dp)
        )
    }

    // Drag handle (only show in non-radio mode)
    if (!isRadioMode) {
        Icon(
            Icons.Filled.DragHandle,
```

```
                    contentDescription = stringResource(R.string.drag_to_reorder),
                    tint = MaterialTheme.colorScheme.onSurfaceVariant.copy(alpha = if (isDragg
                    modifier = Modifier.size(20.dp)
                )
            }
        }
    }
}

// File: java\com\example\holodex\ui\composables\HeroCard.kt
// File: java/com/example/holodex/ui/composables/HeroCard.kt
// (Create this new file)

package com.example.holodex.ui.composables

import androidx.compose.foundation.background
import androidx.compose.foundation.clickable
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.aspectRatio
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.graphics.Brush
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.layout.ContentScale
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.text.style.TextOverflow
import androidx.compose.ui.unit.dp
import coil.compose.AsyncImage
import coil.request.ImageRequest
import com.example.holodex.R
import com.example.holodex.data.model.discovery.SingingStreamShelfItem
import com.example.holodex.util.ThumbnailQuality
import com.example.holodex.util.getYouTubeThumbnailUrl

@Composable
fun HeroCard(
    item: SingingStreamShelfItem,
    onClick: () -> Unit,
    modifier: Modifier = Modifier
) {
    val video = item.video
    val thumbnailUrl = getYouTubeThumbnailUrl(video.id, ThumbnailQuality.MAX).firstOrNull()

    Box(
```

```
    modifier = modifier
        .fillMaxWidth()
        .aspectRatio(16f / 9f)
        .clip(MaterialTheme.shapes.large)
        .clickable(onClick = onClick)
) {
    // Background Image
    AsyncImage(
        model = ImageRequest.Builder(LocalContext.current)
            .data(thumbnailUrl)
            .placeholder(R.drawable.ic_placeholder_image)
            .error(R.drawable.ic_error_image)
            .crossfade(true)
            .build(),
        contentDescription = video.title,
        contentScale = ContentScale.Crop,
        modifier = Modifier.fillMaxSize()
    )

    // Gradient overlay for text readability
    Box(
        modifier = Modifier
            .fillMaxSize()
            .background(
                Brush.verticalGradient(
                    colors = listOf(
                        Color.Transparent,
                        Color.Black.copy(alpha = 0.2f),
                        Color.Black.copy(alpha = 0.8f)
                    ),
                    startY = 300f
                )
            )
    )

    // Text Content
    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp),
        verticalArrangement = Arrangement.Bottom
    ) {
        Text(
            text = video.title,
            style = MaterialTheme.typography.titleLarge,
            color = Color.White,
            fontWeight = FontWeight.Bold,
            maxLines = 2,
            overflow = TextOverflow.Ellipsis
        )
        Spacer(modifier = Modifier.height(4.dp))
        Text(
            text = video.channel.name,
            style = MaterialTheme.typography.bodyMedium,
            color = Color.White.copy(alpha = 0.9f),
```

```
                maxLines = 1,
                overflow = TextOverflow.Ellipsis
            )
        }
    }
}

// File: java\com\example\holodex\ui\composables\HeroCarousel.kt
// File: java/com/example/holodex/ui/composables/HeroCarousel.kt
// (Create this new file)

package com.example.holodex.ui.composables

import androidx.compose.foundation.ExperimentalFoundationApi
import androidx.compose.foundation.background
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.PaddingValues
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.aspectRatio
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.pager.HorizontalPager
import androidx.compose.foundation.pager.rememberPagerState
import androidx.compose.foundation.shape.CircleShape
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.unit.dp
import com.example.holodex.data.model.discovery.SingingStreamShelfItem
import com.example.holodex.viewmodel.state.UiState

@OptIn(ExperimentalFoundationApi::class)
@Composable
fun HeroCarousel(
    title: String,
    uiState: UiState<List<SingingStreamShelfItem>>,
    onItemClicked: (SingingStreamShelfItem) -> Unit,
    modifier: Modifier = Modifier,
) {
    Column(modifier = modifier) {
        // Title (no "Show More" button)
        Text(
            text = title,
            style = MaterialTheme.typography.titleLarge,
            modifier = Modifier.padding(horizontal = 16.dp)
        )
        Spacer(Modifier.height(12.dp))
```

```
when (uiState) {
    is UiState.Loading -> {
        // Show a single large skeleton
        Box(
            modifier = Modifier
                .fillMaxWidth()
                .padding(horizontal = 16.dp)
                .aspectRatio(16f / 9f)
                .clip(MaterialTheme.shapes.large)
                .background(MaterialTheme.colorScheme.surfaceVariant)
        )
    }
    is UiState.Success -> {
        if (uiState.data.isNotEmpty()) {
            val pagerState = rememberPagerState(pageCount = { uiState.data.size })

            HorizontalPager(
                state = pagerState,
                contentPadding = PaddingValues(horizontal = 16.dp),
                pageSpacing = 12.dp,
            ) { pageIndex ->
                HeroCard(
                    item = uiState.data[pageIndex],
                    onClick = { onItemClicked(uiState.data[pageIndex]) }
                )
            }

            // Pager Indicators
            Row(
                Modifier
                    .height(24.dp)
                    .fillMaxWidth(),
                horizontalArrangement = Arrangement.Center,
                verticalAlignment = Alignment.Bottom
            ) {
                repeat(pagerState.pageCount) { iteration ->
                    val color = if (pagerState.currentPage == iteration) MaterialTheme
                    Box(
                        modifier = Modifier
                            .padding(4.dp)
                            .clip(CircleShape)
                            .background(color)
                            .size(8.dp)
                    )
                }
            }
        }
    }
    is UiState.Error -> {
        // You can reuse the error component from CarouselShelf if you extract it
        Text(
            text = uiState.message,
            color = MaterialTheme.colorScheme.error,
            modifier = Modifier.padding(horizontal = 16.dp)
```

```
                )
            }
        }
    }
}

// File: java\com\example\holodex\ui\composables\ItemOptionsMenu.kt
package com.example.holodex.ui.composables

import android.content.Intent
import androidx.compose.foundation.layout.padding
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.automirrored.filled.PlaylistAdd
import androidx.compose.material.icons.automirrored.filled.QueueMusic
import androidx.compose.material.icons.filled.Delete
import androidx.compose.material.icons.filled.Download
import androidx.compose.material.icons.filled.Movie
import androidx.compose.material.icons.filled.Person
import androidx.compose.material.icons.filled.Share
import androidx.compose.material3.DropdownMenu
import androidx.compose.material3.DropdownMenuItem
import androidx.compose.material3.HorizontalDivider
import androidx.compose.material3.Icon
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.Immutable
import androidx.compose.ui.Modifier
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.unit.dp
import com.example.holodex.R

@Composable
fun ItemOptionsMenu(
    state: ItemMenuState,
    actions: ItemMenuActions,
    expanded: Boolean,
    onDismissRequest: () -> Unit
) {
    val context = LocalContext.current
    val onShare = { textToShare: String ->
        val sendIntent: Intent = Intent().apply {
            action = Intent.ACTION_SEND
            putExtra(Intent.EXTRA_TEXT, textToShare)
            type = "text/plain"
        }
        val shareIntent = Intent.createChooser(sendIntent, null)
        context.startActivity(shareIntent)
        onDismissRequest()
    }

    DropdownMenu(expanded = expanded, onDismissRequest = onDismissRequest) {
        DropdownMenuItem(
            text = { Text(stringResource(R.string.action_add_to_queue)) },
            onClick = {
```

```
            actions.onAddToQueue()
            onDismissRequest()
        },
        leadingIcon = { Icon(Icons.AutoMirrored.Filled.QueueMusic, null) }
    )

    DropdownMenuItem(
        text = { Text(stringResource(R.string.action_add_to_playlist_menu)) },
        onClick = {
            actions.onAddToPlaylist()
            onDismissRequest()
        },
        leadingIcon = { Icon(Icons.AutoMirrored.Filled.PlaylistAdd, null) }
    )

    DropdownMenuItem(
        text = { Text(stringResource(R.string.action_share)) },
        onClick = { onShare(state.shareUrl) },
        leadingIcon = { Icon(Icons.Filled.Share, null) }
    )

    if (state.canBeDownloaded) {
        DropdownMenuItem(
            text = { Text(stringResource(R.string.action_download)) },
            onClick = {
                actions.onDownload()
                onDismissRequest()
            },
            leadingIcon = { Icon(Icons.Filled.Download, null) }
        )
    }

    if (state.isDownloaded) {
        DropdownMenuItem(
            text = { Text(stringResource(R.string.action_delete)) },
            onClick = {
                actions.onDelete()
                onDismissRequest()
            },
            leadingIcon = { Icon(Icons.Filled.Delete, null) }
        )
    }

    HorizontalDivider(modifier = Modifier.padding(vertical = 4.dp))

    if (state.isSegment) {
        DropdownMenuItem(
            text = { Text(stringResource(R.string.action_view_video)) },
            onClick = {
                actions.onGoToVideo(state.videoId)
                onDismissRequest()
            },
            leadingIcon = { Icon(Icons.Filled.Movie, null) }
        )
    }
```

```
        DropdownMenuItem(
            text = { Text(stringResource(R.string.action_view_artist)) },
            onClick = {
                actions.onGoToArtist(state.channelId)
                onDismissRequest()
            },
            leadingIcon = { Icon(Icons.Filled.Person, null) },
            enabled = state.channelId.isNotBlank()
        )
    }
}
/**
 * A state holder for the ItemOptionsMenu. It contains all the necessary
 * data to determine the visibility and enabled status of menu items.
 */
@Immutable
data class ItemMenuState(
    val isDownloaded: Boolean,
    val isSegment: Boolean,
    val canBeDownloaded: Boolean,
    val shareUrl: String,
    val videoId: String,
    val channelId: String
)

/**
 * A holder for all the possible actions a user can take from the ItemOptionsMenu.
 * The parent composable is responsible for providing the implementations for these actions.
 */
@Immutable
data class ItemMenuActions(
    val onAddToQueue: () -> Unit,
    val onAddToPlaylist: () -> Unit,
    val onShare: (String) -> Unit,
    val onDownload: () -> Unit,
    val onDelete: () -> Unit,
    val onGoToVideo: (String) -> Unit,
    val onGoToArtist: (String) -> Unit,
)

// File: java\com\example\holodex\ui\composables\Media3PlayerControls.kt
@file:UnstableApi

package com.example.holodex.ui.composables

import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.PauseCircleFilled
import androidx.compose.material.icons.filled.PlayCircleFilled
```

```
import androidx.compose.material.icons.filled.SkipNext
import androidx.compose.material.icons.filled.SkipPrevious
import androidx.compose.material3.Icon
import androidx.compose.material3.IconButton
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Slider
import androidx.compose.material3.SliderDefaults
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableFloatStateOf
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.res.painterResource
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.unit.dp
import androidx.media3.common.Player
import androidx.media3.common.util.UnstableApi
import com.example.holodex.R
import com.example.holodex.playback.domain.model.DomainPlaybackProgress
import com.example.holodex.playback.domain.model.DomainRepeatMode
import com.example.holodex.playback.domain.model.DomainShuffleMode
import com.example.holodex.playback.util.formatDurationSecondsToString

private const val CONTROLS_TAG = "Media3PlayerControls"

/**
 * A self-contained composable that displays a full set of player controls,
 * powered by Media3's Compose state holders.
 *
 * @param player The Media3 Player instance.
 * @param progress The current playback progress, passed from the ViewModel to display on the
 * @param onSeek A lambda to be invoked when the user interacts with the seek bar.
 * @param primaryColor The primary theme color, used for prominent elements like the play butt
 * @param onPrimaryColor The color for icons and text that appear on the primary color, used f
 */
@Composable
fun Media3PlayerControls(
    // The player is now ONLY for reading state for button enabled/disabled status
    player: Player,
    shuffleMode: DomainShuffleMode,
    repeatMode: DomainRepeatMode,
    progress: DomainPlaybackProgress,
    isRadioMode: Boolean,
    onPlayPause: () -> Unit,
    onSkipPrevious: () -> Unit,
    onSkipNext: () -> Unit,
    onToggleShuffle: () -> Unit,
    onToggleRepeat: () -> Unit,
    onSeek: (Long) -> Unit,
```

```
    onScrubbingChange: (Boolean) -> Unit,
    primaryColor: Color,
    onPrimaryColor: Color,
    modifier: Modifier = Modifier
) {
    Column(
        modifier = modifier
            .fillMaxWidth()
            .padding(horizontal = 16.dp, vertical = 8.dp)
    ) {
        PlayerSeekBar(
            progress = progress,
            onSeek = onSeek,
            onScrubbingChange = onScrubbingChange,
            thumbColor = primaryColor,
            activeTrackColor = onPrimaryColor,
            inactiveTrackColor = onPrimaryColor.copy(alpha = 0.3f),
            timeTextColor = onPrimaryColor.copy(alpha = 0.7f)
        )

        Row(
            modifier = Modifier.fillMaxWidth(),
            horizontalArrangement = Arrangement.SpaceEvenly,
            verticalAlignment = Alignment.CenterVertically
        ) {
            // --- CUSTOM SHUFFLE BUTTON ---
            val isShuffleOn = shuffleMode == DomainShuffleMode.ON // Use the provided state
            IconButton(
                onClick = onToggleShuffle,
                enabled = player.isCommandAvailable(Player.COMMAND_SET_SHUFFLE_MODE) && !isRad
            ) {
                Icon(
                    painter = painterResource(id = if (isShuffleOn) R.drawable.ic_shuffle_on_2
                    contentDescription = stringResource(R.string.action_shuffle),
                    modifier = Modifier.size(28.dp),
                    tint = if (isShuffleOn) primaryColor else onPrimaryColor.copy(alpha = 0.6f
                )
            }

            // --- CUSTOM PREVIOUS BUTTON ---
            IconButton(
                onClick = onSkipPrevious,
                enabled = player.isCommandAvailable(Player.COMMAND_SEEK_TO_PREVIOUS_MEDIA_ITEM
            ) {
                Icon(
                    imageVector = Icons.Filled.SkipPrevious,
                    contentDescription = stringResource(R.string.action_previous),
                    modifier = Modifier.size(36.dp),
                    tint = onPrimaryColor
                )
            }

            // --- CUSTOM PLAY/PAUSE BUTTON ---
            IconButton(
                onClick = onPlayPause,
```

```
                enabled = player.isCommandAvailable(Player.COMMAND_PLAY_PAUSE)
            ) {
                Icon(
                    imageVector = if (player.isPlaying) Icons.Filled.PauseCircleFilled else Ic
                    contentDescription = if (player.isPlaying) stringResource(R.string.action_
                        R.string.action_play
                    ),
                    modifier = Modifier.size(64.dp),
                    tint = primaryColor
                )
            }

            // --- CUSTOM NEXT BUTTON ---
            IconButton(
                onClick = onSkipNext,
                enabled = player.isCommandAvailable(Player.COMMAND_SEEK_TO_NEXT_MEDIA_ITEM)
            ) {
                Icon(
                    imageVector = Icons.Filled.SkipNext,
                    contentDescription = stringResource(R.string.action_next),
                    modifier = Modifier.size(36.dp),
                    tint = onPrimaryColor
                )
            }

            // --- CUSTOM REPEAT BUTTON ---
            IconButton(
                onClick = onToggleRepeat,
                enabled = player.isCommandAvailable(Player.COMMAND_SET_REPEAT_MODE) && !isRadi
            ) {
                val iconRes = when (repeatMode) { // Use the provided state
                    DomainRepeatMode.ONE -> R.drawable.ic_repeat_one_24
                    DomainRepeatMode.ALL -> R.drawable.ic_repeat_on_24
                    else -> R.drawable.ic_repeat_off_24
                }
                Icon(
                    painter = painterResource(id = iconRes),
                    contentDescription = stringResource(R.string.action_repeat),
                    modifier = Modifier.size(28.dp),
                    tint = if (repeatMode != DomainRepeatMode.NONE) primaryColor else onPrimar
                        alpha = 0.6f
                    )
                )
            }
        }
    }
}


// ================================================================
// PRIVATE, INTERNAL BUILDING BLOCKS FOR THE CONTROLS
// ================================================================

@Composable
private fun PlayerSeekBar(
```

```
    progress: DomainPlaybackProgress,
    onSeek: (Long) -> Unit,
    onScrubbingChange: (Boolean) -> Unit,
    thumbColor: Color,
    activeTrackColor: Color,
    inactiveTrackColor: Color,
    timeTextColor: Color
) {
    var sliderPosition by remember(progress.positionSec) { mutableFloatStateOf(progress.positi
    var isUserScrubbing by remember { mutableStateOf(false) }

    LaunchedEffect(isUserScrubbing) {
        onScrubbingChange(isUserScrubbing)
    }

    Column(Modifier.fillMaxWidth()) {
        Slider(
            value = if (isUserScrubbing) sliderPosition else progress.positionSec.toFloat(),
            onValueChange = {
                isUserScrubbing = true
                sliderPosition = it
            },
            onValueChangeFinished = {
                onSeek(sliderPosition.toLong())
                isUserScrubbing = false
            },
            valueRange = 0f..(progress.durationSec.toFloat().coerceAtLeast(1f)),
            modifier = Modifier.fillMaxWidth(),
            colors = SliderDefaults.colors(
                thumbColor = thumbColor,
                activeTrackColor = activeTrackColor,
                inactiveTrackColor = inactiveTrackColor
            )
        )
        Row(
            modifier = Modifier
                .fillMaxWidth()
                .padding(horizontal = 4.dp),
            horizontalArrangement = Arrangement.SpaceBetween
        ) {
            Text(
                text = formatDurationSecondsToString(if (isUserScrubbing) sliderPosition.toLon
                style = MaterialTheme.typography.bodySmall,
                color = timeTextColor
            )
            Text(
                text = formatDurationSecondsToString(progress.durationSec),
                style = MaterialTheme.typography.bodySmall,
                color = timeTextColor
            )
        }
    }
}
```

```
// File: java\com\example\holodex\ui\composables\MiniPlayerWithProgressBar.kt
// File: java/com/example/holodex/ui/composables/MiniPlayerWithProgressBar.kt

package com.example.holodex.ui.composables

import androidx.compose.animation.AnimatedVisibility
import androidx.compose.animation.fadeOut
import androidx.compose.animation.shrinkVertically
import androidx.compose.foundation.background
import androidx.compose.foundation.clickable
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.IntrinsicSize
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.layout.width
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.Pause
import androidx.compose.material.icons.filled.PlayArrow
import androidx.compose.material.icons.filled.SkipNext
import androidx.compose.material3.ExperimentalMaterial3Api
import androidx.compose.material3.Icon
import androidx.compose.material3.IconButton
import androidx.compose.material3.LinearProgressIndicator
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Surface
import androidx.compose.material3.SwipeToDismissBox
import androidx.compose.material3.SwipeToDismissBoxValue
import androidx.compose.material3.Text
import androidx.compose.material3.rememberSwipeToDismissBoxState
import androidx.compose.material3.surfaceColorAtElevation
import androidx.compose.runtime.Composable
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.layout.ContentScale
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.text.style.TextOverflow
import androidx.compose.ui.unit.dp
import androidx.lifecycle.compose.collectAsStateWithLifecycle
import coil.compose.AsyncImage
import coil.request.ImageRequest
import com.example.holodex.R
import com.example.holodex.util.ThumbnailQuality
```

```
import com.example.holodex.util.generateArtworkUrlList
import com.example.holodex.viewmodel.PlaybackViewModel
import com.example.holodex.viewmodel.rememberFullPlayerLoadingState
import com.example.holodex.viewmodel.rememberIsPlayingState
import com.example.holodex.viewmodel.rememberMiniPlayerArtistState
import com.example.holodex.viewmodel.rememberMiniPlayerProgressState
import com.example.holodex.viewmodel.rememberMiniPlayerQueueStateForButton
import com.example.holodex.viewmodel.rememberMiniPlayerTitleState
import kotlinx.coroutines.delay

@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun MiniPlayerWithProgressBar(
    playbackViewModel: PlaybackViewModel,
    modifier: Modifier = Modifier,
    onTapped: () -> Unit = {}
) {
    val uiState by playbackViewModel
        .uiState
        .collectAsStateWithLifecycle()
    val currentItem = uiState.currentItem

    val title by rememberMiniPlayerTitleState(playbackViewModel.uiState)
    val artist by rememberMiniPlayerArtistState(playbackViewModel.uiState)
    val isPlaying by rememberIsPlayingState(playbackViewModel.uiState)
    val progressFraction by rememberMiniPlayerProgressState(playbackViewModel.uiState)
    val queueStatePair by rememberMiniPlayerQueueStateForButton(playbackViewModel.uiState)
    // FIX 2: Deconstruct to ignore the unused variable
    val (_, canSkipNext) = queueStatePair
    val isLoading by rememberFullPlayerLoadingState(playbackViewModel.uiState)

    // FIX 3: Simplify this check. If title is not null, an item exists.
    val currentItemExists = title != null

    var showPlayer by remember { mutableStateOf(true) }

    val dismissState = rememberSwipeToDismissBoxState(
        confirmValueChange = {
            if (it == SwipeToDismissBoxValue.StartToEnd || it == SwipeToDismissBoxValue.EndToS
                showPlayer = false
                true
            } else {
                false
            }
        }
    )

    LaunchedEffect(showPlayer) {
        if (!showPlayer) {
            delay(300)
            playbackViewModel.clearCurrentQueue()
        }
    }

    LaunchedEffect(currentItem?.id) {
```

```
    if (currentItem != null && !showPlayer) {
        showPlayer = true
        dismissState.reset()
    }
}

if (!currentItemExists && !isLoading) {
    Spacer(modifier = modifier.height(0.dp))
    return
}

AnimatedVisibility(
    visible = showPlayer,
    exit = shrinkVertically() + fadeOut()
) {
    SwipeToDismissBox(
        state = dismissState,
        backgroundContent = {},
        modifier = modifier
    ) {
        Surface(
            modifier = Modifier
                .fillMaxWidth()
                .height(IntrinsicSize.Min)
                .clickable(onClick = onTapped, enabled = currentItemExists),
            color = MaterialTheme.colorScheme.surfaceColorAtElevation(3.dp),
            tonalElevation = 3.dp,
            shadowElevation = 3.dp
        ) {
            Column {
                Row(
                    modifier = Modifier
                        .fillMaxWidth()
                        .padding(
                            start = 8.dp,
                            end = 4.dp,
                            top = 8.dp,
                            bottom = 8.dp
                        ),
                    verticalAlignment = Alignment.CenterVertically
                ) {
                    val miniPlayerArtworkUrls = remember(currentItem) {
                        generateArtworkUrlList(currentItem, ThumbnailQuality.HIGH)
                    }

                    AsyncImage(
                        model = ImageRequest.Builder(LocalContext.current)
                            .data(miniPlayerArtworkUrls.firstOrNull())
                            .placeholder(R.drawable.ic_default_album_art_placeholder)
                            .error(R.drawable.ic_error_image)
                            .crossfade(true)
                            .build(),
                        contentDescription = stringResource(R.string.content_desc_album_ar
                        contentScale = ContentScale.Crop,
                        modifier = Modifier
```

```
            .size(48.dp)
            .clip(MaterialTheme.shapes.small)
            .background(MaterialTheme.colorScheme.surfaceVariant)
    )

    Spacer(modifier = Modifier.width(12.dp))

    Column(
        modifier = Modifier.weight(1f),
        verticalArrangement = Arrangement.Center
    ) {
        Text(
            text = title ?: stringResource(R.string.loading_track),
            style = MaterialTheme.typography.titleMedium,
            maxLines = 1,
            overflow = TextOverflow.Ellipsis,
            color = MaterialTheme.colorScheme.onSurface
        )
        // FIX 1: Use `let` to create a local, smart-casted variable
        artist?.let { artistText ->
            Text(
                text = artistText,
                style = MaterialTheme.typography.bodyMedium,
                maxLines = 1,
                overflow = TextOverflow.Ellipsis,
                color = MaterialTheme.colorScheme.onSurfaceVariant
            )
        }
    }

    Spacer(modifier = Modifier.width(4.dp))

    IconButton(
        onClick = { playbackViewModel.togglePlayPause() },
        enabled = currentItemExists && !isLoading
    ) {
        Icon(
            imageVector = if (isPlaying) Icons.Filled.Pause else Icons.Fil
            contentDescription = if (isPlaying) stringResource(R.string.ac
            modifier = Modifier.size(36.dp),
            tint = MaterialTheme.colorScheme.primary
        )
    }

    IconButton(
        onClick = { playbackViewModel.skipToNext() },
        enabled = canSkipNext && !isLoading
    ) {
        Icon(
            imageVector = Icons.Filled.SkipNext,
            contentDescription = stringResource(R.string.action_next),
            modifier = Modifier.size(36.dp),
            tint = MaterialTheme.colorScheme.onSurfaceVariant
        )
    }
```

```
                    }

                    if (isLoading && !currentItemExists) {
                        LinearProgressIndicator(modifier = Modifier.fillMaxWidth().height(2.dp
                    } else if (currentItemExists) {
                        LinearProgressIndicator(
                            progress = { progressFraction },
                            modifier = Modifier.fillMaxWidth().height(2.dp),
                            color = MaterialTheme.colorScheme.primary,
                            trackColor = MaterialTheme.colorScheme.surfaceVariant.copy(alpha =
                        )
                    }
                }
            }
        }
    }
}

// File: java\com\example\holodex\ui\composables\PlayerBackground.kt
package com.example.holodex.ui.composables

import androidx.compose.animation.animateColorAsState
import androidx.compose.animation.core.tween
import androidx.compose.foundation.background
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.runtime.remember
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.blur
import androidx.compose.ui.graphics.Brush
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.graphics.ColorFilter
import androidx.compose.ui.graphics.ColorMatrix
import androidx.compose.ui.layout.ContentScale
import androidx.compose.ui.unit.dp
import coil.compose.AsyncImage

/**
 * A highly optimized, reusable composable that displays a blurred and themed background
 * based on album artwork. It's designed for performance by minimizing overdraw and
 * caching expensive graphics objects.
 *
 * @param artworkUri The URL of the artwork to display in the background.
 * @param dynamicColor The dominant color extracted from the artwork, used for the gradient ov
 * @param modifier The modifier to be applied to this composable.
 */
@Composable
fun SimpleProcessedBackground(
    artworkUri: String?,
    dynamicColor: Color,
    modifier: Modifier = Modifier,
    blurRadius: Int = 80,
    saturation: Float = 0.5f,
```

```
    darkenFactor: Float = 0.7f
) {
    val animatedPrimaryColor by animateColorAsState(
        targetValue = dynamicColor,
        label = "animated_primary_color_background",
        animationSpec = tween(1200)
    )

    val colorFilter = remember(saturation, darkenFactor) {
        ColorFilter.colorMatrix(
            ColorMatrix().apply {
                setToSaturation(saturation)
                val values = floatArrayOf(
                    darkenFactor, 0f, 0f, 0f, 0f,
                    0f, darkenFactor, 0f, 0f, 0f,
                    0f, 0f, darkenFactor, 0f, 0f,
                    0f, 0f, 0f, 1f, 0f
                )
                timesAssign(ColorMatrix(values))
            }
        )
    }

    val gradientBrush = remember(animatedPrimaryColor) {
        Brush.verticalGradient(
            colors = listOf(
                animatedPrimaryColor.copy(alpha = 0.2f),
                animatedPrimaryColor.copy(alpha = 0.4f),
                Color.Black.copy(alpha = 0.7f)
            )
        )
    }

    Box(modifier = modifier.fillMaxSize()) {
        AsyncImage(
            model = artworkUri,
            contentDescription = "Background artwork",
            contentScale = ContentScale.Crop,
            modifier = Modifier
                .fillMaxSize()
                .blur(radius = blurRadius.dp),
            colorFilter = colorFilter
        )
        Box(
            modifier = Modifier
                .fillMaxSize()
                .background(gradientBrush)
        )
    }
}

// File: java\com\example\holodex\ui\composables\PlaylistArtwork.kt
package com.example.holodex.ui.composables

import androidx.compose.foundation.background
```

```
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.aspectRatio
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.shape.CircleShape
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.Podcasts
import androidx.compose.material3.Icon
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Surface
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.remember
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.draw.shadow
import androidx.compose.ui.graphics.Brush
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.layout.ContentScale
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.text.style.TextAlign
import androidx.compose.ui.text.style.TextOverflow
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import coil.compose.AsyncImage
import coil.request.ImageRequest
import com.example.holodex.R
import com.example.holodex.data.model.discovery.PlaylistStub
import com.example.holodex.util.ArtworkResolver
import com.example.holodex.util.PlaylistFormatter
import java.util.Locale
import kotlin.math.max

@Composable
fun PlaylistArtwork(
    playlist: PlaylistStub,
    modifier: Modifier = Modifier
) {
    val context = LocalContext.current

    val artworkUrl = remember(playlist.id) {
        ArtworkResolver.getPlaylistArtworkUrl(playlist)
    }
    val (type, title) = remember(playlist.id, playlist.title) {
        val formattedTitle = PlaylistFormatter.getDisplayTitle(playlist, context) { englishNam
            japaneseName?.takeIf { it.isNotBlank() } ?: englishName
        }
        val playlistType = if (playlist.id.startsWith(":")) {
            playlist.id.substringBefore('[')
```

```
        } else {
            "ugp"
        }
        playlistType to formattedTitle
    }

    Surface(
        modifier = modifier
            .fillMaxWidth()
            .aspectRatio(1f)
            .clip(MaterialTheme.shapes.medium),
        shadowElevation = 2.dp
    ) {
        when (type) {
            ":artist", ":hot" -> RadioTextArt(
                titleText = title,
                imageUrl = artworkUrl,
            )
            ":dailyrandom", ":weekly", ":mv", ":latest" -> {
                val lastSpaceIndex = title.lastIndexOf(' ')
                val (typeText, titleText) = if (lastSpaceIndex > 0 && title.length > lastSpace
                    title.substring(0, lastSpaceIndex) to title.substring(lastSpaceIndex + 1)
                } else {
                    title.substringBefore(": ") to title.substringAfter(": ", title)
                }
                StackedTextArt(
                    typeText = typeText,
                    titleText = titleText,
                    imageUrl = artworkUrl
                )
            }
            else -> OverlayTextArt(
                titleText = title,
                imageUrl = artworkUrl
            )
        }
    }
}

@Composable
private fun OverlayTextArt(
    titleText: String,
    imageUrl: String?,
    modifier: Modifier = Modifier
) {
    Box(modifier = modifier.fillMaxSize(), contentAlignment = Alignment.BottomStart) {
        AsyncImage(
            model = ImageRequest.Builder(LocalContext.current)
                .data(imageUrl)
                .placeholder(R.drawable.ic_placeholder_image)
                .error(R.drawable.ic_error_image)
                .crossfade(true)
                .build(),
            contentDescription = titleText,
            contentScale = ContentScale.Crop,
```

```
            modifier = Modifier.fillMaxSize()
        )
        Box(
            modifier = Modifier
                .fillMaxSize()
                .background(
                    Brush.verticalGradient(
                        colors = listOf(Color.Transparent, Color.Black.copy(alpha = 0.8f)),
                        startY = 150f
                    )
                )
        )
        Text(
            text = titleText,
            style = MaterialTheme.typography.bodyLarge,
            fontWeight = FontWeight.Bold,
            color = Color.White,
            maxLines = 3,
            overflow = TextOverflow.Ellipsis,
            modifier = Modifier.padding(12.dp)
        )
    }
}

@Composable
private fun RadioTextArt(
    titleText: String,
    imageUrl: String?,
    modifier: Modifier = Modifier
) {
    Box(
        modifier = modifier
            .fillMaxSize()
            .background(
                Brush.radialGradient(
                    colors = listOf(
                        MaterialTheme.colorScheme.onSurface.copy(alpha = 0.2f),
                        MaterialTheme.colorScheme.surfaceVariant.copy(alpha = 0.1f),
                    ),
                    radius = 250f
                )
            )
            .background(MaterialTheme.colorScheme.surfaceVariant),
        contentAlignment = Alignment.Center
    ) {
        Icon(
            imageVector = Icons.Default.Podcasts,
            contentDescription = null,
            modifier = Modifier
                .fillMaxSize(0.9f)
                .align(Alignment.Center),
            tint = MaterialTheme.colorScheme.onSurface.copy(alpha = 0.1f)
        )
        Column(
            horizontalAlignment = Alignment.CenterHorizontally,
```

```
            verticalArrangement = Arrangement.spacedBy(4.dp),
            modifier = Modifier.padding(8.dp)
        ) {
            AsyncImage(
                model = ImageRequest.Builder(LocalContext.current)
                    .data(imageUrl)
                    .placeholder(R.drawable.ic_placeholder_image)
                    .error(R.drawable.ic_error_image)
                    .crossfade(true)
                    .build(),
                contentDescription = titleText,
                contentScale = ContentScale.Crop,
                modifier = Modifier
                    .fillMaxWidth(0.6f)
                    .aspectRatio(1f)
                    .shadow(elevation = 8.dp, shape = CircleShape)
                    .clip(CircleShape)
            )
            Text(
                text = titleText,
                style = MaterialTheme.typography.titleSmall,
                textAlign = TextAlign.Center,
                maxLines = 1,
                overflow = TextOverflow.Ellipsis,
                modifier = Modifier.padding(top = 4.dp)
            )
            Text(
                text = stringResource(id = R.string.sgp_radio_type),
                style = MaterialTheme.typography.bodySmall,
                color = MaterialTheme.colorScheme.onSurfaceVariant,
                textAlign = TextAlign.Center
            )
        }
    }
}

@Composable
private fun StackedTextArt(
    typeText: String,
    titleText: String,
    imageUrl: String?,
    modifier: Modifier = Modifier
) {
    val adjFontSize = max(14, (18 - (titleText.length / 15))).sp

    Column(modifier = modifier.fillMaxSize()) {
        Column(
            modifier = Modifier
                .fillMaxWidth()
                .weight(0.4f)
                .background(MaterialTheme.colorScheme.primaryContainer)
                .padding(horizontal = 12.dp, vertical = 4.dp),
            verticalArrangement = Arrangement.Center
        ) {
            Text(
```

```
                text = typeText.toUpperCase(Locale.getDefault()),
                style = MaterialTheme.typography.labelMedium,
                color = MaterialTheme.colorScheme.onPrimaryContainer.copy(alpha = 0.7f),
            )
            Text(
                text = titleText,
                style = MaterialTheme.typography.titleMedium.copy(fontSize = adjFontSize),
                color = MaterialTheme.colorScheme.onPrimaryContainer,
                maxLines = 2,
                overflow = TextOverflow.Ellipsis,
                fontWeight = FontWeight.Bold
            )
        }
        AsyncImage(
            model = ImageRequest.Builder(LocalContext.current)
                .data(imageUrl)
                .placeholder(R.drawable.ic_placeholder_image)
                .error(R.drawable.ic_error_image)
                .crossfade(true)
                .build(),
            contentDescription = titleText,
            contentScale = ContentScale.Crop,
            modifier = Modifier
                .fillMaxWidth()
                .weight(0.6f)
        )
    }
}

// File: java\com\example\holodex\ui\composables\PlaylistCard.kt
package com.example.holodex.ui.composables

import androidx.compose.foundation.clickable
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.width
import androidx.compose.material3.Card
import androidx.compose.material3.CardDefaults
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.remember
import androidx.compose.ui.Modifier
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.text.style.TextOverflow
import androidx.compose.ui.unit.dp
import com.example.holodex.data.model.discovery.PlaylistStub
import com.example.holodex.util.PlaylistFormatter

@Composable
fun PlaylistCard(
    playlist: PlaylistStub,
    onPlaylistClicked: (PlaylistStub) -> Unit,
    modifier: Modifier = Modifier
) {
```

```
    val context = LocalContext.current

    // --- START OF IMPLEMENTATION ---
    val (type, displayTitle, displayDescription) = remember(playlist) {
        val playlistType = if (playlist.id.startsWith(":")) playlist.id.substringBefore('[') e
        val title = PlaylistFormatter.getDisplayTitle(playlist, context) { en, jp -> jp?.takeI
        val description = PlaylistFormatter.getDisplayDescription(playlist, context) { en, jp
        Triple(playlistType, title, description)
    }

    val textToShow = when (type) {
        ":artist", ":hot" -> displayDescription ?: displayTitle
        ":dailyrandom", ":weekly", ":mv", ":latest" -> displayDescription ?: displayTitle
        else -> displayTitle // For UGP and others, show the title
    }
    // --- END OF IMPLEMENTATION ---

    Card(
        modifier = modifier
            .width(140.dp)
            .clickable { onPlaylistClicked(playlist) },
        colors = CardDefaults.cardColors(containerColor = MaterialTheme.colorScheme.surface)
    ) {
        Column {
            PlaylistArtwork(
                playlist = playlist,
                modifier = Modifier
            )
            Text(
                text = textToShow,
                style = MaterialTheme.typography.bodyMedium,
                maxLines = 2,
                overflow = TextOverflow.Ellipsis,
                modifier = Modifier.padding(8.dp)
            )
        }
    }
}

// File: java\com\example\holodex\ui\composables\PlaylistManagementDialogs.kt
package com.example.holodex.ui.composables

import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.lifecycle.compose.collectAsStateWithLifecycle
import com.example.holodex.ui.dialogs.CreatePlaylistDialog
import com.example.holodex.ui.dialogs.SelectPlaylistDialog
import com.example.holodex.viewmodel.PlaylistManagementViewModel

@Composable
fun PlaylistManagementDialogs(
    playlistManagementViewModel: PlaylistManagementViewModel
) {
    // Using standard lifecycle collection for these booleans
    val showSelectPlaylistDialog by playlistManagementViewModel.showSelectPlaylistDialog.colle
```

```
    val userPlaylistsForDialog by playlistManagementViewModel.userPlaylists.collectAsStateWith
    val showCreatePlaylistDialog by playlistManagementViewModel.showCreatePlaylistDialog.colle

    if (showSelectPlaylistDialog) {
        SelectPlaylistDialog(
            playlists = userPlaylistsForDialog,
            onDismissRequest = { playlistManagementViewModel.cancelAddToPlaylistFlow() },
            onPlaylistSelected = { playlist -> playlistManagementViewModel.addItemToExistingPl
            onCreateNewPlaylistClicked = { playlistManagementViewModel.handleCreateNewPlaylist
        )
    }
    if (showCreatePlaylistDialog) {
        CreatePlaylistDialog(
            onDismissRequest = { playlistManagementViewModel.cancelAddToPlaylistFlow() },
            onCreatePlaylist = { name, desc -> playlistManagementViewModel.confirmCreatePlayli
        )
    }
}

// File: java\com\example\holodex\ui\composables\StateDisplayComposables.kt
package com.example.holodex.ui.composables

import androidx.compose.foundation.background
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.layout.width
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.ErrorOutline
import androidx.compose.material.icons.filled.MusicOff
import androidx.compose.material.icons.filled.Refresh
import androidx.compose.material3.Button
import androidx.compose.material3.ButtonDefaults
import androidx.compose.material3.CircularProgressIndicator
import androidx.compose.material3.Icon
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.text.style.TextAlign
import androidx.compose.ui.unit.dp
import com.example.holodex.R
```

```
@Composable
fun LoadingState(message: String, modifier: Modifier = Modifier) {
    Box(modifier = modifier.fillMaxSize(), contentAlignment = Alignment.Center) {
        Column(
            horizontalAlignment = Alignment.CenterHorizontally,
            verticalArrangement = Arrangement.Center
        ) {
            CircularProgressIndicator()
            Spacer(modifier = Modifier.height(16.dp))
            Text(
                text = message,
                style = MaterialTheme.typography.bodyLarge,
                textAlign = TextAlign.Center
            )
        }
    }
}

@Composable
fun LoadingSkeleton(modifier: Modifier = Modifier, itemCount: Int = 10) {
    LazyColumn(modifier = modifier) {
        items(itemCount) {
            Row(
                modifier = Modifier
                    .fillMaxWidth()
                    .padding(horizontal = 16.dp, vertical = 8.dp),
                verticalAlignment = Alignment.CenterVertically
            ) {
                Box(
                    modifier = Modifier
                        .size(56.dp)
                        .clip(RoundedCornerShape(8.dp))
                        .background(MaterialTheme.colorScheme.surfaceVariant.copy(alpha = 0.3f
                )
                Spacer(modifier = Modifier.width(12.dp))
                Column(verticalArrangement = Arrangement.spacedBy(6.dp)) {
                    Box(
                        modifier = Modifier
                            .fillMaxWidth(0.7f)
                            .height(18.dp)
                            .clip(RoundedCornerShape(4.dp))
                            .background(MaterialTheme.colorScheme.surfaceVariant.copy(alpha =
                    )
                    Box(
                        modifier = Modifier
                            .fillMaxWidth(0.5f)
                            .height(14.dp)
                            .clip(RoundedCornerShape(4.dp))
                            .background(MaterialTheme.colorScheme.surfaceVariant.copy(alpha =
                    )
                }
            }
        }
    }
}
```

```
@Composable
fun EmptyState(message: String, onRefresh: () -> Unit, modifier: Modifier = Modifier) {
    Box(
        modifier = modifier.fillMaxSize().padding(16.dp),
        contentAlignment = Alignment.Center
    ) {
        Column(
            horizontalAlignment = Alignment.CenterHorizontally,
            verticalArrangement = Arrangement.Center
        ) {
            Icon(
                imageVector = Icons.Filled.MusicOff,
                contentDescription = null,
                modifier = Modifier.size(64.dp),
                tint = MaterialTheme.colorScheme.onSurfaceVariant.copy(alpha = 0.4f)
            )
            Spacer(modifier = Modifier.height(16.dp))
            Text(
                text = message,
                color = MaterialTheme.colorScheme.onSurfaceVariant,
                textAlign = TextAlign.Center,
                style = MaterialTheme.typography.bodyLarge
            )
            Spacer(modifier = Modifier.height(16.dp))
            Button(onClick = onRefresh) {
                Icon(Icons.Filled.Refresh, contentDescription = stringResource(R.string.action
                Spacer(modifier = Modifier.size(ButtonDefaults.IconSpacing))
                Text(stringResource(R.string.action_refresh))
            }
        }
    }
}

@Composable
fun ErrorStateWithRetry(message: String, onRetry: () -> Unit, modifier: Modifier = Modifier) {
    Box(
        modifier = modifier.fillMaxSize().padding(16.dp),
        contentAlignment = Alignment.Center
    ) {
        Column(
            horizontalAlignment = Alignment.CenterHorizontally,
            verticalArrangement = Arrangement.Center
        ) {
            Icon(
                imageVector = Icons.Filled.ErrorOutline,
                contentDescription = null,
                modifier = Modifier.size(64.dp),
                tint = MaterialTheme.colorScheme.error
            )
            Spacer(modifier = Modifier.height(16.dp))
            Text(
                text = message,
                color = MaterialTheme.colorScheme.error,
                textAlign = TextAlign.Center,
```

```
                style = MaterialTheme.typography.bodyLarge
            )
            Spacer(modifier = Modifier.height(16.dp))
            Button(onClick = onRetry) {
                Icon(Icons.Filled.Refresh, contentDescription = stringResource(R.string.action
                Spacer(modifier = Modifier.size(ButtonDefaults.IconSpacing))
                Text(stringResource(R.string.action_retry))
            }
        }
    }
}

// File: java\com\example\holodex\ui\composables\UnifiedGridItem.kt
package com.example.holodex.ui.composables

import androidx.compose.foundation.background
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.aspectRatio
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.layout.width
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.CloudDone
import androidx.compose.material3.Card
import androidx.compose.material3.ExperimentalMaterial3Api
import androidx.compose.material3.Icon
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableIntStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.graphics.Brush
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.layout.ContentScale
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.text.style.TextOverflow
import androidx.compose.ui.unit.dp
import coil.compose.AsyncImage
import coil.request.ImageRequest
import com.example.holodex.R
import com.example.holodex.viewmodel.UnifiedDisplayItem

/**
```

```
 * A universal, reusable composable for displaying any music-related item in a grid or carouse
 * It intelligently displays badges and context based on the properties of the UnifiedDisplayI
 *
 * @param item The canonical UnifiedDisplayItem containing all necessary data for display.
 * @param onClick The action to perform when the card is clicked.
 * @param modifier The modifier to be applied to the Card.
 */
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun UnifiedGridItem(
    item: UnifiedDisplayItem,
    onClick: () -> Unit,
    modifier: Modifier = Modifier,
) {
    var currentUrlIndex by remember(item.artworkUrls) { mutableIntStateOf(0) }

    Card(
        onClick = onClick,
        modifier = modifier.width(140.dp)
    ) {
        Column {
            Box(contentAlignment = Alignment.BottomStart) {
                // Main Artwork Image
                AsyncImage(
                    model = ImageRequest.Builder(LocalContext.current)
                        .data(item.artworkUrls.getOrNull(currentUrlIndex))
                        .placeholder(R.drawable.ic_placeholder_image)
                        .error(R.drawable.ic_error_image)
                        .crossfade(true)
                        .build(),
                    onError = {
                        // If the primary URL fails, try the next one in the prioritized list
                        if (currentUrlIndex < item.artworkUrls.lastIndex) {
                            currentUrlIndex++
                        }
                    },
                    contentDescription = stringResource(R.string.content_desc_album_art_for, i
                    contentScale = ContentScale.Crop,
                    modifier = Modifier
                        .fillMaxWidth()
                        .aspectRatio(1f)
                        .clip(MaterialTheme.shapes.medium)
                )

                // Gradient scrim for text readability
                Box(
                    modifier = Modifier
                        .matchParentSize()
                        .background(
                            Brush.verticalGradient(
                                colors = listOf(Color.Transparent, Color.Black.copy(alpha = 0.
                                startY = 100f // Start gradient lower down
                            )
                        )
                )
```

```
                // Badges and Duration, overlaid on the artwork
                Row(
                    modifier = Modifier
                        .fillMaxWidth()
                        .padding(6.dp),
                    horizontalArrangement = Arrangement.End,
                    verticalAlignment = Alignment.CenterVertically
                ) {
                    if (item.isDownloaded) {
                        Icon(
                            imageVector = Icons.Filled.CloudDone,
                            contentDescription = stringResource(R.string.content_description_d
                            tint = Color.White,
                            modifier = Modifier
                                .size(16.dp)
                                .padding(end = 4.dp)
                        )
                    }
                    Text(
                        text = item.durationText,
                        style = MaterialTheme.typography.labelSmall,
                        color = Color.White,
                        fontWeight = FontWeight.Bold,
                        modifier = Modifier
                            .background(Color.Black.copy(alpha = 0.5f), RoundedCornerShape(4.d
                            .padding(horizontal = 4.dp, vertical = 2.dp)
                    )
                }
            }

            // Text content below the artwork
            Column(
                modifier = Modifier.padding(8.dp),
                verticalArrangement = Arrangement.spacedBy(2.dp)
            ) {
                Text(
                    text = item.title,
                    style = MaterialTheme.typography.titleSmall,
                    maxLines = 2,
                    overflow = TextOverflow.Ellipsis
                )
                Text(
                    text = item.artistText,
                    style = MaterialTheme.typography.bodySmall,
                    maxLines = 1,
                    overflow = TextOverflow.Ellipsis,
                    color = MaterialTheme.colorScheme.onSurfaceVariant
                )
            }
        }
    }
}

// File: java\com\example\holodex\ui\composables\UnifiedListItem.kt
```

```
// File: java/com/example/holodex/ui/composables/UnifiedListItem.kt
package com.example.holodex.ui.composables

import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.layout.width
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.CloudDone
import androidx.compose.material.icons.filled.Delete
import androidx.compose.material.icons.filled.DragHandle
import androidx.compose.material.icons.filled.Favorite
import androidx.compose.material.icons.filled.FavoriteBorder
import androidx.compose.material.icons.filled.MoreVert
import androidx.compose.material3.Card
import androidx.compose.material3.CardDefaults
import androidx.compose.material3.ExperimentalMaterial3Api
import androidx.compose.material3.Icon
import androidx.compose.material3.IconButton
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableIntStateOf
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.hapticfeedback.HapticFeedbackType
import androidx.compose.ui.layout.ContentScale
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.platform.LocalHapticFeedback
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.text.style.TextOverflow
import androidx.compose.ui.unit.dp
import androidx.hilt.navigation.compose.hiltViewModel
import androidx.media3.common.util.UnstableApi
import androidx.navigation.NavController
import coil.compose.AsyncImage
import coil.request.ImageRequest
import com.example.holodex.R
import com.example.holodex.ui.AppDestinations
import com.example.holodex.viewmodel.DownloadsViewModel
import com.example.holodex.viewmodel.FavoritesViewModel
import com.example.holodex.viewmodel.PlaylistManagementViewModel
import com.example.holodex.viewmodel.UnifiedDisplayItem
```

```
import com.example.holodex.viewmodel.VideoDetailsViewModel
import com.example.holodex.viewmodel.VideoListViewModel
import com.example.holodex.viewmodel.mappers.toPlaybackItem
import org.orbitmvi.orbit.compose.collectAsState

@UnstableApi
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun UnifiedListItem(
    item: UnifiedDisplayItem,
    onItemClicked: () -> Unit,
    navController: NavController,
    videoListViewModel: VideoListViewModel,
    favoritesViewModel: FavoritesViewModel,
    playlistManagementViewModel: PlaylistManagementViewModel,
    modifier: Modifier = Modifier,
    isEditing: Boolean = false,
    onRemoveClicked: () -> Unit = {},
    dragHandleModifier: Modifier = Modifier,
    isExternal: Boolean = false
) {
    val haptic = LocalHapticFeedback.current
    var currentUrlIndex by remember(item.artworkUrls) { mutableIntStateOf(0) }
    var showOptionsMenu by remember { mutableStateOf(false) }
    val favoritesState by favoritesViewModel.collectAsState()


    val isItemLiked = remember(item.playbackItemId, favoritesState.likedItemsMap) {
        favoritesState.likedItemsMap.containsKey(item.playbackItemId)
    }
    val videoDetailsViewModel: VideoDetailsViewModel = hiltViewModel()
    val downloadsViewModel: DownloadsViewModel = hiltViewModel()

    val menuState = remember(item) {
        ItemMenuState(
            isDownloaded = item.isDownloaded,
            isSegment = item.isSegment,
            canBeDownloaded = item.isSegment && !item.isDownloaded,
            shareUrl = if (item.isSegment && item.songStartSec != null) {
                "https://music.holodex.net/watch/${item.videoId}/${item.songStartSec}"
            } else {
                "https://music.holodex.net/watch/${item.videoId}"
            },
            videoId = item.videoId,
            channelId = item.channelId
        )
    }

    val menuActions = remember(item) {
        ItemMenuActions(
            onAddToQueue = { videoListViewModel.addVideoOrItsSegmentsToQueue(item.toPlaybackIt
            onAddToPlaylist = {
                playlistManagementViewModel.prepareItemForPlaylistAddition(item)
            },
            onShare = { /* Handled internally by the menu now */ },
```

```
        onDownload = { videoDetailsViewModel.requestDownloadForSongFromPlaybackItem(item.t
        onDelete = { downloadsViewModel.deleteDownload(item.playbackItemId) },
        onGoToVideo = { videoId -> navController.navigate(AppDestinations.videoDetailRoute
        onGoToArtist = { channelId -> navController.navigate("channel_details/$channelId")

    )
}

Card(
    onClick = {
        haptic.performHapticFeedback(HapticFeedbackType.TextHandleMove)
        onItemClicked()
    },
    modifier = modifier
        .fillMaxWidth()
        .padding(horizontal = 16.dp, vertical = 6.dp),
    elevation = CardDefaults.cardElevation(defaultElevation = 2.dp),
    shape = MaterialTheme.shapes.medium
) {
    Row(
        modifier = Modifier
            .fillMaxWidth()
            .padding(start = 12.dp, top = 8.dp, bottom = 8.dp, end = 4.dp),
        verticalAlignment = Alignment.CenterVertically
    ) {
        AsyncImage(
            model = ImageRequest.Builder(LocalContext.current)
                .data(item.artworkUrls.getOrNull(currentUrlIndex))
                .placeholder(R.drawable.ic_placeholder_image)
                .error(R.drawable.ic_error_image)
                .crossfade(true)
                .build(),
            onError = {
                if (currentUrlIndex < item.artworkUrls.lastIndex) {
                    currentUrlIndex++
                }
            },
            contentDescription = stringResource(R.string.content_desc_channel_thumbnail),
            contentScale = ContentScale.Crop,
            modifier = Modifier
                .size(56.dp)
                .clip(RoundedCornerShape(8.dp))
        )

        Spacer(modifier = Modifier.width(12.dp))

        Column(
            modifier = Modifier
                .weight(1f)
                .padding(end = 4.dp),
            verticalArrangement = Arrangement.spacedBy(3.dp)
        ) {
            Row(verticalAlignment = Alignment.CenterVertically) {
                Text(
                    text = item.title,
```

```
                style = MaterialTheme.typography.titleSmall,
                fontWeight = FontWeight.Medium,
                maxLines = 2,
                overflow = TextOverflow.Ellipsis,
                color = MaterialTheme.colorScheme.onSurface,
                modifier = Modifier.weight(1f, fill = false)
            )
            if (item.isSegment && item.isDownloaded) {
                Spacer(Modifier.width(6.dp))
                Icon(
                    Icons.Filled.CloudDone,
                    contentDescription = "Downloaded",
                    tint = MaterialTheme.colorScheme.primary,
                    modifier = Modifier.size(16.dp)
                )
            }
        }

        Text(
            text = item.artistText,
            style = MaterialTheme.typography.bodyMedium,
            maxLines = 1,
            overflow = TextOverflow.Ellipsis,
            color = MaterialTheme.colorScheme.onSurfaceVariant
        )

        Row(verticalAlignment = Alignment.CenterVertically) {
            if (item.durationText.isNotEmpty()) {
                Text(
                    text = item.durationText,
                    style = MaterialTheme.typography.bodySmall,
                    color = MaterialTheme.colorScheme.onSurfaceVariant
                )
            }
            if (!item.isSegment) {
                item.songCount?.let { count ->
                    if (count > 0) {
                        if (item.durationText.isNotEmpty()) {
                            Text(
                                " ? ",
                                style = MaterialTheme.typography.bodySmall,
                                color = MaterialTheme.colorScheme.onSurfaceVariant
                            )
                        }
                        Text(
                            stringResource(R.string.song_count_format, count),
                            style = MaterialTheme.typography.bodySmall,
                            color = MaterialTheme.colorScheme.onSurfaceVariant
                        )
                    }
                }
            }
        }
    }
}
```

```
            if (isEditing) {
                IconButton(onClick = onRemoveClicked) {
                    Icon(
                        Icons.Default.Delete,
                        contentDescription = stringResource(R.string.action_remove_from_playli
                        tint = MaterialTheme.colorScheme.error
                    )
                }
                Box(modifier = dragHandleModifier) {
                    Icon(
                        Icons.Default.DragHandle,
                        contentDescription = stringResource(R.string.drag_to_reorder)
                    )
                }
            } else {
                IconButton(
                    onClick = {
                        val playbackItem = item.toPlaybackItem().copy(isExternal = isExternal)
                        favoritesViewModel.toggleLike(playbackItem)
                        haptic.performHapticFeedback(HapticFeedbackType.LongPress)
                    },
                    modifier = Modifier.size(40.dp)
                ) {
                    Icon(
                        imageVector = if (isItemLiked) Icons.Filled.Favorite else Icons.Filled
                        contentDescription = if (isItemLiked) stringResource(R.string.action_u
                        tint = if (isItemLiked) MaterialTheme.colorScheme.primary else Materia
                        modifier = Modifier.size(24.dp)
                    )
                }

                Box {
                    IconButton(onClick = { showOptionsMenu = true }, modifier = Modifier.size(
                        Icon(Icons.Filled.MoreVert, stringResource(R.string.action_more_option
                    }
                    ItemOptionsMenu(
                        state = menuState,
                        actions = menuActions,
                        expanded = showOptionsMenu,
                        onDismissRequest = { showOptionsMenu = false }
                    )
                }
            }
        }
    }
}

// File: java\com\example\holodex\ui\composables\sheets\BrowseFiltersSheet.kt
@file:OptIn(androidx.media3.common.util.UnstableApi::class)
package com.example.holodex.ui.composables.sheets// File: java\com\example\holodex\ui\composab
// ... (imports)import com.example.holodex.viewmodel.state.SongSegmentFilterMode
import androidx.compose.foundation.clickable
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
```

```
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.navigationBarsPadding
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.width
import androidx.compose.foundation.rememberScrollState
import androidx.compose.foundation.verticalScroll
import androidx.compose.material3.Button
import androidx.compose.material3.DropdownMenuItem
import androidx.compose.material3.ExperimentalMaterial3Api
import androidx.compose.material3.ExposedDropdownMenuBox
import androidx.compose.material3.ExposedDropdownMenuDefaults
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.OutlinedTextField
import androidx.compose.material3.RadioButton
import androidx.compose.material3.Text
import androidx.compose.material3.TextButton
import androidx.compose.runtime.Composable
import androidx.compose.runtime.derivedStateOf
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.unit.dp
import androidx.hilt.navigation.compose.hiltViewModel
import androidx.lifecycle.compose.collectAsStateWithLifecycle
import com.example.holodex.R
import com.example.holodex.viewmodel.VideoListViewModel
import com.example.holodex.viewmodel.state.BrowseFilterState
import com.example.holodex.viewmodel.state.SortOrder
import com.example.holodex.viewmodel.state.VideoSortField
import com.example.holodex.viewmodel.state.ViewTypePreset


@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun BrowseFiltersSheet(
    initialFilters: BrowseFilterState,
    onFiltersApplied: (BrowseFilterState) -> Unit,
    onDismiss: () -> Unit,
    videoListViewModel: VideoListViewModel = hiltViewModel()
) {
    var tempFilters by remember { mutableStateOf(initialFilters) }

    val organizationsForDropdown by videoListViewModel.availableOrganizations.collectAsStateWi

    val viewTypePresetOptions = videoListViewModel.browseScreenCategories

    var viewPresetExpanded by remember { mutableStateOf(false) }
    var orgExpanded by remember { mutableStateOf(false) }
    var sortFieldExpanded by remember { mutableStateOf(false) }
```

```
Column(
    modifier = Modifier
        .fillMaxWidth()
        .navigationBarsPadding()
        .padding(16.dp)
        .verticalScroll(rememberScrollState())
) {
    Text("Filter & Sort Music Streams", style = MaterialTheme.typography.titleLarge, modif

    FilterSectionHeader("View As")
    val currentViewPresetDisplay by remember(tempFilters, organizationsForDropdown) {
        derivedStateOf {
            val orgDisplayPart = if (tempFilters.selectedOrganization != null) {
                val orgName = organizationsForDropdown.find { it.second == tempFilters.sel
                if (orgName != "All Vtubers") " - $orgName" else ""
            } else ""

            val segmentDisplayPart = if (tempFilters.selectedViewPreset == ViewTypePreset.
                tempFilters.songSegmentFilterMode.displayNameSuffix ?: ""
            } else {
                ""
            }
            "${tempFilters.selectedViewPreset.defaultDisplayName}$orgDisplayPart$segmentDi
        }
    }

    ExposedDropdownMenuBox(
        expanded = viewPresetExpanded,
        onExpandedChange = { viewPresetExpanded = it },
        modifier = Modifier.fillMaxWidth()
    ) {
        OutlinedTextField(
            value = currentViewPresetDisplay,
            onValueChange = {}, readOnly = true, label = { Text("View Type") },
            trailingIcon = { ExposedDropdownMenuDefaults.TrailingIcon(expanded = viewPrese
            modifier = Modifier.menuAnchor().fillMaxWidth()
        )
        ExposedDropdownMenu(expanded = viewPresetExpanded, onDismissRequest = { viewPreset
            viewTypePresetOptions.forEach { (displayName, presetBrowseFilterStateFromVM) -
                DropdownMenuItem(
                    text = { Text(displayName) },
                    onClick = {
                        val currentOrgApiVal = tempFilters.selectedOrganization
                        val currentPrimaryTopic = tempFilters.selectedPrimaryTopic // Pres
                        val currentSortField = tempFilters.sortField
                        val currentSortOrder = tempFilters.sortOrder

                        var newFilterState = BrowseFilterState.create(
                            preset = presetBrowseFilterStateFromVM.selectedViewPreset,
                            songFilterMode = presetBrowseFilterStateFromVM.songSegmentFilt
                            organization = currentOrgApiVal,
                            primaryTopic = currentPrimaryTopic,
                            // Maintain existing sort if it makes sense for the new preset
                            sortFieldOverride = if (presetBrowseFilterStateFromVM.selected
```

```
                        sortOrderOverride = if (presetBrowseFilterStateFromVM.selected
                    )

                    // Post-adjustment: if the chosen preset changed the sort field to
                    // force it to a default compatible sort for the new preset.
                    if (newFilterState.selectedViewPreset == ViewTypePreset.LATEST_STR
                        if (newFilterState.sortField == VideoSortField.START_SCHEDULED
                            newFilterState = newFilterState.copy(
                                sortField = VideoSortField.AVAILABLE_AT, // Default to
                                sortOrder = SortOrder.DESC
                            )
                        }
                    } else if (newFilterState.selectedViewPreset == ViewTypePreset.UPC
                        if (newFilterState.sortField != VideoSortField.START_SCHEDULED
                            newFilterState = newFilterState.copy(
                                sortField = VideoSortField.START_SCHEDULED,
                                sortOrder = SortOrder.ASC // Upcoming usually sorted b
                            )
                        }
                    }

                    tempFilters = newFilterState
                    viewPresetExpanded = false
                }
            )
        }
    }
}
Spacer(Modifier.height(16.dp))

FilterSectionHeader("Organization")
val currentOrgDisplay = organizationsForDropdown.find { it.second == tempFilters.selec
ExposedDropdownMenuBox(expanded = orgExpanded, onExpandedChange = { orgExpanded = it }
    OutlinedTextField(
        value = currentOrgDisplay, onValueChange = {}, readOnly = true, label = { Text
        trailingIcon = { ExposedDropdownMenuDefaults.TrailingIcon(expanded = orgExpand
        modifier = Modifier.menuAnchor().fillMaxWidth()
    )
    ExposedDropdownMenu(expanded = orgExpanded, onDismissRequest = { orgExpanded = fal
        // --- FIX: Iterate over the collected list ---
        organizationsForDropdown.forEach { (name, value) ->
            DropdownMenuItem(text = { Text(name) }, onClick = {
                tempFilters = BrowseFilterState.create(
                    preset = tempFilters.selectedViewPreset,
                    songFilterMode = tempFilters.songSegmentFilterMode,
                    organization = value,
                    primaryTopic = tempFilters.selectedPrimaryTopic,
                    sortFieldOverride = tempFilters.sortField,
                    sortOrderOverride = tempFilters.sortOrder,
                )
                orgExpanded = false
            })
        }
    }
}
```

```
Spacer(Modifier.height(16.dp))

FilterSectionHeader("Sort By")
ExposedDropdownMenuBox(expanded = sortFieldExpanded, onExpandedChange = { sortFieldExp
    OutlinedTextField(
        value = tempFilters.sortField.displayName, onValueChange = {}, readOnly = true
        trailingIcon = { ExposedDropdownMenuDefaults.TrailingIcon(expanded = sortField
        modifier = Modifier.menuAnchor().fillMaxWidth()
    )
    ExposedDropdownMenu(expanded = sortFieldExpanded, onDismissRequest = { sortFieldEx
        VideoSortField.entries.forEach { field ->
            val isApplicable = when (tempFilters.selectedViewPreset) {
                ViewTypePreset.UPCOMING_STREAMS -> {
                    field == VideoSortField.START_SCHEDULED ||
                            field == VideoSortField.LIVE_VIEWERS ||
                            field == VideoSortField.TITLE ||
                            field == VideoSortField.PUBLISHED_AT
                }
                ViewTypePreset.LATEST_STREAMS -> {
                    field != VideoSortField.START_SCHEDULED && field != VideoSortField
                }
            }

            if (isApplicable) {
                DropdownMenuItem(text = { Text(field.displayName) }, onClick = {
                    var newSortOrder = tempFilters.sortOrder
                    // Automatically set default sort order based on field
                    if (field == VideoSortField.AVAILABLE_AT || field == VideoSortFiel
                        newSortOrder = SortOrder.DESC
                    } else if (field == VideoSortField.START_SCHEDULED) {
                        newSortOrder = SortOrder.ASC
                    } else if (field == VideoSortField.LIVE_VIEWERS) {
                        newSortOrder = SortOrder.DESC
                    }

                    tempFilters = tempFilters.copy(sortField = field, sortOrder = newS
                    sortFieldExpanded = false
                })
            }
        }
    }
}
Spacer(Modifier.height(8.dp))

Row(verticalAlignment = Alignment.CenterVertically, modifier = Modifier.fillMaxWidth()
    SortOrder.entries.forEach { order ->
        Row(Modifier.clickable { tempFilters = tempFilters.copy(sortOrder = order) }.p
            RadioButton(selected = tempFilters.sortOrder == order, onClick = { tempFil
            Text(order.displayName, style = MaterialTheme.typography.bodyLarge, modifi
        }
    }
}
Spacer(Modifier.height(24.dp))

Row(modifier = Modifier.fillMaxWidth(), horizontalArrangement = Arrangement.End) {
```

```
            TextButton(onClick = onDismiss) { Text(stringResource(id = R.string.cancel)) }
            Spacer(Modifier.width(8.dp))
            Button(onClick = {
                onFiltersApplied(tempFilters)
            }) { Text(stringResource(id = R.string.apply_filters_button)) }
        }
    }
}

@Composable
fun FilterSectionHeader(title: String) {
    Text(
        text = title,
        style = MaterialTheme.typography.titleSmall,
        modifier = Modifier.padding(top = 12.dp, bottom = 8.dp)
    )
}

// File: java\com\example\holodex\ui\dialogs\AddExternalChannelDialog.kt
// File: java/com/example/holodex/ui/dialogs/AddExternalChannelDialog.kt
package com.example.holodex.ui.dialogs

import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.PaddingValues
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.heightIn
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.foundation.shape.CircleShape
import androidx.compose.foundation.text.KeyboardActions
import androidx.compose.foundation.text.KeyboardOptions
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.Add
import androidx.compose.material.icons.filled.Clear
import androidx.compose.material.icons.filled.Search
import androidx.compose.material3.Button
import androidx.compose.material3.Card
import androidx.compose.material3.CircularProgressIndicator
import androidx.compose.material3.ExperimentalMaterial3Api
import androidx.compose.material3.HorizontalDivider
import androidx.compose.material3.Icon
import androidx.compose.material3.IconButton
import androidx.compose.material3.ListItem
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.OutlinedTextField
import androidx.compose.material3.Text
import androidx.compose.material3.TextButton
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.ui.Alignment
```

```
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.layout.ContentScale
import androidx.compose.ui.platform.LocalFocusManager
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.text.input.ImeAction
import androidx.compose.ui.text.style.TextAlign
import androidx.compose.ui.unit.dp
import androidx.compose.ui.window.Dialog
import androidx.hilt.navigation.compose.hiltViewModel
import androidx.lifecycle.compose.collectAsStateWithLifecycle
import coil.compose.AsyncImage
import com.example.holodex.R
import com.example.holodex.data.model.ChannelSearchResult
import com.example.holodex.viewmodel.ExternalChannelViewModel
import com.example.holodex.viewmodel.state.UiState

@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun AddExternalChannelDialog(
    onDismissRequest: () -> Unit,
    viewModel: ExternalChannelViewModel = hiltViewModel()
) {
    val searchQuery by viewModel.searchQuery.collectAsStateWithLifecycle()
    val searchState by viewModel.searchState.collectAsStateWithLifecycle()
    val isAdding by viewModel.isAdding.collectAsStateWithLifecycle()
    val focusManager = LocalFocusManager.current

    Dialog(onDismissRequest = onDismissRequest) {
        Card(
            modifier = Modifier.fillMaxWidth().heightIn(max = 600.dp),
            shape = MaterialTheme.shapes.large
        ) {
            Column {
                Text(
                    text = "Add External Channel",
                    style = MaterialTheme.typography.titleLarge,
                    modifier = Modifier.padding(16.dp)
                )
                OutlinedTextField(
                    value = searchQuery,
                    onValueChange = { viewModel.onSearchQueryChanged(it) },
                    modifier = Modifier.fillMaxWidth().padding(horizontal = 16.dp),
                    placeholder = { Text("Search YouTube for a channel...") },
                    leadingIcon = { Icon(Icons.Default.Search, null) },
                    trailingIcon = {
                        if (searchQuery.isNotEmpty()) {
                            IconButton(onClick = { viewModel.onSearchQueryChanged("") }) {
                                Icon(Icons.Default.Clear, "Clear search")
                            }
                        }
                    },
                    singleLine = true,
                    keyboardOptions = KeyboardOptions(imeAction = ImeAction.Search),
                    keyboardActions = KeyboardActions(onSearch = { focusManager.clearFocus() }
```

```
                )

                Box(
                    modifier = Modifier.weight(1f).fillMaxWidth(),
                    contentAlignment = Alignment.Center
                ) {
                    when (val state = searchState) {
                        is UiState.Loading -> CircularProgressIndicator()
                        is UiState.Error -> Text(state.message, color = MaterialTheme.colorSch
                        is UiState.Success -> {
                            if (state.data.isEmpty() && searchQuery.length > 2) {
                                Text("No channels found.", modifier = Modifier.padding(16.dp))
                            } else {
                                LazyColumn(contentPadding = PaddingValues(16.dp)) {
                                    items(state.data, key = { it.channelId }) { channel ->
                                        ChannelSearchResultItem(
                                            channel = channel,
                                            isAdding = isAdding.contains(channel.channelId),
                                            onAddClicked = { viewModel.addChannel(channel) }
                                        )
                                    }
                                }
                            }
                        }
                    }
                }

                HorizontalDivider()
                Row(
                    modifier = Modifier.fillMaxWidth().padding(horizontal = 16.dp, vertical =
                    horizontalArrangement = Arrangement.End
                ) {
                    TextButton(onClick = onDismissRequest) {
                        Text(stringResource(R.string.cancel))
                    }
                }
            }
        }
    }
}

@Composable
private fun ChannelSearchResultItem(
    channel: ChannelSearchResult,
    isAdding: Boolean,
    onAddClicked: () -> Unit
) {
    ListItem(
        headlineContent = { Text(channel.name, maxLines = 1, overflow = androidx.compose.ui.te
        supportingContent = { channel.subscriberCount?.let { Text(it) } },
        leadingContent = {
            AsyncImage(
                model = channel.thumbnailUrl,
                contentDescription = channel.name,
                modifier = Modifier.size(40.dp).clip(CircleShape),
```

```
                contentScale = ContentScale.Crop
            )
        },
        trailingContent = {
            Button(onClick = onAddClicked, enabled = !isAdding) {
                if (isAdding) {
                    CircularProgressIndicator(modifier = Modifier.size(18.dp), strokeWidth = 2
                } else {
                    Icon(Icons.Default.Add, null)
                }
            }
        }
    )
}

// File: java\com\example\holodex\ui\dialogs\CreatePlaylistDialog.kt
package com.example.holodex.ui.dialogs

import androidx.compose.foundation.layout.*
import androidx.compose.material3.*
import androidx.compose.runtime.*
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.unit.dp
import androidx.compose.ui.window.Dialog
import com.example.holodex.R // Assuming strings are in R.string

@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun CreatePlaylistDialog(
    onDismissRequest: () -> Unit,
    onCreatePlaylist: (name: String, description: String?) -> Unit
) {
    var playlistName by remember { mutableStateOf("") }
    var playlistDescription by remember { mutableStateOf("") }

    Dialog(onDismissRequest = onDismissRequest) {
        Card(
            modifier = Modifier
                .fillMaxWidth()
                .padding(16.dp),
            shape = MaterialTheme.shapes.large
        ) {
            Column(
                modifier = Modifier.padding(16.dp),
                horizontalAlignment = Alignment.CenterHorizontally
            ) {
                Text(
                    text = stringResource(R.string.dialog_title_create_playlist),
                    style = MaterialTheme.typography.titleLarge,
                    modifier = Modifier.padding(bottom = 16.dp)
                )
                OutlinedTextField(
                    value = playlistName,
```

```
                    onValueChange = { playlistName = it },
                    label = { Text(stringResource(R.string.hint_playlist_name)) },
                    singleLine = true,
                    modifier = Modifier.fillMaxWidth()
                )
                Spacer(Modifier.height(8.dp))
                OutlinedTextField(
                    value = playlistDescription,
                    onValueChange = { playlistDescription = it },
                    label = { Text(stringResource(R.string.hint_playlist_description_optional)
                    modifier = Modifier.fillMaxWidth(),
                    maxLines = 3
                )
                Spacer(Modifier.height(24.dp))
                Row(
                    modifier = Modifier.fillMaxWidth(),
                    horizontalArrangement = Arrangement.End
                ) {
                    TextButton(onClick = onDismissRequest) {
                        Text(stringResource(R.string.cancel))
                    }
                    Spacer(Modifier.width(8.dp))
                    Button(
                        onClick = {
                            onCreatePlaylist(playlistName, playlistDescription.ifBlank { null
                        },
                        enabled = playlistName.isNotBlank()
                    ) {
                        Text(stringResource(R.string.create))
                    }
                }
            }
        }
    }
}

// File: java\com\example\holodex\ui\dialogs\SelectPlaylistDialog.kt
package com.example.holodex.ui.dialogs

import androidx.compose.foundation.clickable
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.heightIn
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.Add
import androidx.compose.material3.Card
import androidx.compose.material3.ExperimentalMaterial3Api
import androidx.compose.material3.HorizontalDivider
import androidx.compose.material3.Icon
import androidx.compose.material3.ListItem
```

```
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Text
import androidx.compose.material3.TextButton
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.unit.dp
import androidx.compose.ui.window.Dialog
import com.example.holodex.R
import com.example.holodex.data.db.PlaylistEntity

@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun SelectPlaylistDialog(
    playlists: List<PlaylistEntity>,
    onDismissRequest: () -> Unit,
    onPlaylistSelected: (PlaylistEntity) -> Unit,
    onCreateNewPlaylistClicked: () -> Unit
) {
    Dialog(onDismissRequest = onDismissRequest) {
        Card(
            modifier = Modifier.fillMaxWidth().padding(16.dp),
            shape = MaterialTheme.shapes.large
        ) {
            Column {
                Text(
                    text = stringResource(R.string.dialog_title_add_to_playlist),
                    style = MaterialTheme.typography.titleLarge,
                    modifier = Modifier.padding(16.dp)
                )
                HorizontalDivider()
                LazyColumn(modifier = Modifier.heightIn(max = 240.dp)) { // Limit height
                    items(playlists, key = { it.playlistId }) { playlist ->
                        ListItem(
                            headlineContent = { Text(playlist.name ?: "Untitled Playlist") },
                            modifier = Modifier.clickable { onPlaylistSelected(playlist) }
                        )
                        HorizontalDivider()
                    }
                    item {
                        ListItem(
                            headlineContent = { Text(stringResource(R.string.action_create_new
                            leadingContent = { Icon(Icons.Filled.Add, contentDescription = nul
                            modifier = Modifier.clickable(onClick = onCreateNewPlaylistClicked
                        )
                    }
                }
                HorizontalDivider()
                Row(
                    modifier = Modifier.fillMaxWidth().padding(horizontal = 16.dp, vertical =
                    horizontalArrangement = Arrangement.End
                ) {
                    TextButton(onClick = onDismissRequest) {
                        Text(stringResource(R.string.cancel))
                    }
```

```
                }
            }
        }
    }
}

// File: java\com\example\holodex\ui\screens\ChannelScreen.kt
package com.example.holodex.ui.screens

import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.PaddingValues
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.shape.CircleShape
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.automirrored.filled.ArrowBack
import androidx.compose.material.icons.filled.Favorite
import androidx.compose.material.icons.filled.FavoriteBorder
import androidx.compose.material3.Button
import androidx.compose.material3.ButtonDefaults
import androidx.compose.material3.CircularProgressIndicator
import androidx.compose.material3.ExperimentalMaterial3Api
import androidx.compose.material3.Icon
import androidx.compose.material3.IconButton
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.OutlinedButton
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Text
import androidx.compose.material3.TextButton
import androidx.compose.material3.TopAppBar
import androidx.compose.material3.TopAppBarDefaults
import androidx.compose.runtime.Composable
import androidx.compose.runtime.derivedStateOf
import androidx.compose.runtime.getValue
import androidx.compose.runtime.remember
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.layout.ContentScale
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.platform.LocalUriHandler
import androidx.compose.ui.res.painterResource
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.unit.dp
import androidx.hilt.navigation.compose.hiltViewModel
```

```
import androidx.lifecycle.compose.collectAsStateWithLifecycle
import androidx.media3.common.util.UnstableApi
import androidx.navigation.NavController
import coil.compose.AsyncImage
import coil.request.ImageRequest
import com.example.holodex.R
import com.example.holodex.data.db.ExternalChannelEntity
import com.example.holodex.data.db.FavoriteChannelEntity
import com.example.holodex.data.model.discovery.ChannelDetails
import com.example.holodex.data.model.discovery.DiscoveryChannel
import com.example.holodex.data.model.discovery.SingingStreamShelfItem
import com.example.holodex.ui.AppDestinations
import com.example.holodex.ui.composables.CarouselShelf
import com.example.holodex.ui.composables.ChannelCard
import com.example.holodex.ui.composables.SimpleProcessedBackground
import com.example.holodex.ui.composables.UnifiedGridItem
import com.example.holodex.util.ThumbnailQuality
import com.example.holodex.util.findActivity
import com.example.holodex.util.getYouTubeThumbnailUrl
import com.example.holodex.viewmodel.ChannelDetailsViewModel
import com.example.holodex.viewmodel.DiscoveryViewModel
import com.example.holodex.viewmodel.FavoritesViewModel
import com.example.holodex.viewmodel.UnifiedDisplayItem
import com.example.holodex.viewmodel.VideoListViewModel
import com.example.holodex.viewmodel.VideoListViewModel.MusicCategoryType
import com.example.holodex.viewmodel.mappers.toUnifiedDisplayItem
import com.example.holodex.viewmodel.state.UiState
import org.orbitmvi.orbit.compose.collectAsState

@UnstableApi
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun ChannelScreen(
    navController: NavController,
    onNavigateUp: () -> Unit
) {
    val channelViewModel: ChannelDetailsViewModel = hiltViewModel()
    val favoritesViewModel: FavoritesViewModel = hiltViewModel()
    val discoveryViewModel: DiscoveryViewModel = hiltViewModel()
    val videoListViewModel: VideoListViewModel = hiltViewModel(findActivity())

    val detailsState by channelViewModel.channelDetailsState.collectAsStateWithLifecycle()
    val discoveryState by channelViewModel.discoveryState.collectAsStateWithLifecycle()
    val popularSongsState by channelViewModel.popularSongsState.collectAsStateWithLifecycle()
    val favoritesState by favoritesViewModel.collectAsState()
    val dynamicTheme by channelViewModel.dynamicTheme.collectAsStateWithLifecycle()

    val backgroundImageUrl by remember(discoveryState, detailsState) {
        derivedStateOf {
            val detailsData = (detailsState as? UiState.Success)?.data
            detailsData?.bannerUrl?.takeIf { it.isNotBlank() }
                ?: (discoveryState as? UiState.Success)?.data?.recentSingingStreams?.firstOrNu
                    getYouTubeThumbnailUrl(it, ThumbnailQuality.MAX).firstOrNull()
                }
        }
```

```
}


Box(modifier = Modifier.fillMaxSize()) {
    SimpleProcessedBackground(
        artworkUri = backgroundImageUrl,
        dynamicColor = dynamicTheme.primary
    )

    Scaffold(
        topBar = { TopAppBar(title = {}, navigationIcon = { IconButton(onClick = onNavigat
        containerColor = Color.Transparent
    ) { paddingValues ->
        LazyColumn(
            modifier = Modifier.padding(paddingValues).fillMaxSize(),
            contentPadding = PaddingValues(bottom = 80.dp),
            verticalArrangement = Arrangement.spacedBy(24.dp)
        ) {
            item {
                when(val state = detailsState) {
                    is UiState.Success -> ChannelHeader(
                        details = state.data,
                        isFavorited = favoritesState.favoriteChannels.any {
                            (it is FavoriteChannelEntity && it.id == state.data.id) ||
                                    (it is ExternalChannelEntity && it.channelId == state.
                        },
                        onFavoriteClicked = { favoritesViewModel.toggleFavoriteChannelByDe
                    )
                    is UiState.Loading -> Box(modifier = Modifier.height(200.dp).fillMaxWi
                    is UiState.Error -> Text(text = state.message, color = MaterialTheme.c
                }
            }

            item {
                CarouselShelf<UnifiedDisplayItem>(
                    title = "Popular",
                    uiState = popularSongsState,
                    actionContent = {
                        TextButton(onClick = { navController.navigate(AppDestinations.full
                            Text(stringResource(id = R.string.action_show_more))
                        }
                    },
                    itemContent = { item ->
                        UnifiedGridItem(item = item, onClick = { discoveryViewModel.playUn
                    }
                )
            }
            item {
                val recentStreamsUiState: UiState<List<SingingStreamShelfItem>> =
                    remember(discoveryState) {
                        when (val state = discoveryState) {
                            is UiState.Success -> UiState.Success(
                                state.data.recentSingingStreams ?: emptyList()
                            )
```

```
                    is UiState.Error -> UiState.Error(state.message)
                    is UiState.Loading -> UiState.Loading
                }
            }


            CarouselShelf<SingingStreamShelfItem>(
                title = "Latest Streams",
                uiState = recentStreamsUiState,
                actionContent = {
                    TextButton(onClick = {
                        videoListViewModel.setBrowseContextAndNavigate(channelId = cha
                        navController.navigate(AppDestinations.HOME_ROUTE)
                    }) {
                        Text(stringResource(id = R.string.action_show_more))
                    }
                },
                itemContent = { item ->
                    val shell = item.video.toUnifiedDisplayItem(false, emptySet())
                    UnifiedGridItem(
                        item = shell,
                        onClick = {
                            navController.navigate(
                                AppDestinations.videoDetailRoute(item.video.id)
                            )
                        })
                }
            )
        }

        item {
            val otherChannelsUiState: UiState<List<DiscoveryChannel>> = remember(disco
                when (val state = discoveryState) {
                    is UiState.Success -> UiState.Success(state.data.channels ?: empty
                    is UiState.Error -> UiState.Error(state.message)
                    is UiState.Loading -> UiState.Loading
                }
            }
            val orgName = (detailsState as? UiState.Success)?.data?.org ?: "Organizati


            CarouselShelf<DiscoveryChannel>(
                title = "Discover More from $orgName",
                uiState = otherChannelsUiState,
                actionContent = {
                    TextButton(onClick = { /* TODO: Navigate to a full channels list f
                        Text(stringResource(id = R.string.action_show_more))
                    }
                },
                itemContent = { channel ->
                    ChannelCard(channel = channel, onChannelClicked = { navController.
                }
            )
        }
    }
```

```
        }
    }
}

@Composable
private fun ChannelHeader(
    details: ChannelDetails,
    isFavorited: Boolean,
    onFavoriteClicked: () -> Unit
) {
    val uriHandler = LocalUriHandler.current

    Column(
        modifier = Modifier.fillMaxWidth().padding(horizontal = 16.dp),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.spacedBy(12.dp)
    ) {
        AsyncImage(
            model = ImageRequest.Builder(LocalContext.current).data(details.photoUrl).crossfad
            contentDescription = "Channel Avatar",
            modifier = Modifier.size(96.dp).clip(CircleShape),
            contentScale = ContentScale.Crop
        )
        Column(horizontalAlignment = Alignment.CenterHorizontally) {
            Text(details.englishName ?: details.name, style = MaterialTheme.typography.headlin
            details.org?.let { Text(it, style = MaterialTheme.typography.titleMedium, color =
        }
        Row(horizontalArrangement = Arrangement.spacedBy(8.dp)) {
            Button(onClick = onFavoriteClicked) {
                Icon(if (isFavorited) Icons.Default.Favorite else Icons.Default.FavoriteBorder
                Spacer(Modifier.size(ButtonDefaults.IconSpacing))
                Text(if (isFavorited) "Favorited" else "Favorite")
            }
            OutlinedButton(onClick = { uriHandler.openUri("https://youtube.com/channel/${detai
                Icon(painterResource(R.drawable.youtube), null)
            }
            details.twitter?.let {
                OutlinedButton(onClick = { uriHandler.openUri("https://twitter.com/${it}") })
                    Icon(painterResource(R.drawable.twitter), null)
                }
            }
        }
    }
}

// File: java\com\example\holodex\ui\screens\DiscoveryScreen.kt
// File: java/com/example/holodex/ui/screens/DiscoveryScreen.kt

package com.example.holodex.ui.screens

import android.widget.Toast
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.PaddingValues
import androidx.compose.foundation.layout.Spacer
```

```
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.automirrored.filled.QueueMusic
import androidx.compose.material.icons.filled.ArrowDropDown
import androidx.compose.material3.ButtonDefaults
import androidx.compose.material3.DropdownMenu
import androidx.compose.material3.DropdownMenuItem
import androidx.compose.material3.ExperimentalMaterial3Api
import androidx.compose.material3.Icon
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Text
import androidx.compose.material3.TextButton
import androidx.compose.material3.TopAppBar
import androidx.compose.runtime.Composable
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Modifier
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.unit.dp
import androidx.hilt.navigation.compose.hiltViewModel
import androidx.lifecycle.compose.collectAsStateWithLifecycle
import androidx.media3.common.util.UnstableApi
import androidx.navigation.NavController
import com.example.holodex.R
import com.example.holodex.auth.AuthViewModel
import com.example.holodex.data.model.discovery.DiscoveryChannel
import com.example.holodex.data.model.discovery.PlaylistStub
import com.example.holodex.data.model.discovery.SingingStreamShelfItem
import com.example.holodex.ui.AppDestinations
import com.example.holodex.ui.composables.CarouselShelf
import com.example.holodex.ui.composables.ChannelCard
import com.example.holodex.ui.composables.HeroCarousel
import com.example.holodex.ui.composables.PlaylistCard
import com.example.holodex.ui.composables.UnifiedGridItem
import com.example.holodex.util.findActivity
import com.example.holodex.viewmodel.DiscoveryViewModel
import com.example.holodex.viewmodel.ShelfType
import com.example.holodex.viewmodel.UnifiedDisplayItem
import com.example.holodex.viewmodel.VideoListViewModel
import com.example.holodex.viewmodel.VideoListViewModel.MusicCategoryType
import com.example.holodex.viewmodel.mappers.toUnifiedDisplayItem
import com.example.holodex.viewmodel.state.UiState
import kotlinx.coroutines.flow.collectLatest

@androidx.annotation.OptIn(UnstableApi::class)
@OptIn(ExperimentalMaterial3Api::class)
@Composable
```

```
fun DiscoveryScreen(
    navController: NavController,
) {
    val discoveryViewModel: DiscoveryViewModel = hiltViewModel()
    val authViewModel: AuthViewModel = hiltViewModel()
    val videoListViewModel: VideoListViewModel = hiltViewModel(findActivity())

    val uiState by discoveryViewModel.uiState.collectAsStateWithLifecycle()
    val authState by authViewModel.authState.collectAsStateWithLifecycle()
    val selectedOrg by videoListViewModel.selectedOrganization.collectAsStateWithLifecycle()
    val availableOrganizations by videoListViewModel.availableOrganizations.collectAsStateWith
    val context = LocalContext.current

    var showOrgMenu by remember { mutableStateOf(false) }

    LaunchedEffect(authState, selectedOrg) {
        discoveryViewModel.loadDiscoveryContent(selectedOrg, authState)
    }

    LaunchedEffect(Unit) {
        discoveryViewModel.transientMessage.collectLatest { message ->
            Toast.makeText(context, message, Toast.LENGTH_SHORT).show()
        }
    }

    Scaffold(
        topBar = {
            TopAppBar(
                title = { Text(stringResource(R.string.bottom_nav_discover)) },
                actions = {
                    Box {
                        TextButton(onClick = { showOrgMenu = true }) {
                            Text(selectedOrg)
                            Icon(
                                Icons.Default.ArrowDropDown,
                                contentDescription = "Select Organization"
                            )
                        }
                        DropdownMenu(
                            expanded = showOrgMenu,
                            onDismissRequest = { showOrgMenu = false }
                        ) {
                            // --- START OF MODIFICATION ---
                            // Use the dynamic list from the ViewModel
                            availableOrganizations.forEach { (name, value) ->
                                if (value != null) {
                                    DropdownMenuItem(
                                        text = { Text(name) },
                                        onClick = {
                                            videoListViewModel.setOrganization(value)
                                            showOrgMenu = false
                                        }
                                    )
                                }
                            }
```

```
                        // --- END OF MODIFICATION ---
                    }
                }
            }
        )
    }
) { paddingValues ->
    LazyColumn(
        modifier = Modifier.padding(paddingValues).fillMaxSize(),
        contentPadding = PaddingValues(vertical = 16.dp),
        verticalArrangement = Arrangement.spacedBy(24.dp)
    ) {
        items(uiState.shelfOrder, key = { it.name }) { shelfType ->
            val shelfState = uiState.shelves[shelfType] ?: UiState.Loading

            when (shelfType) {
                ShelfType.RECENT_STREAMS -> {
                    // Use the new HeroCarousel for this specific shelf
                    @Suppress("UNCHECKED_CAST")
                    HeroCarousel(
                        title = shelfType.toTitle(selectedOrg),
                        uiState = shelfState as UiState<List<SingingStreamShelfItem>>,
                        onItemClicked = { item ->
                            navController.navigate(AppDestinations.videoDetailRoute(item.v
                        }
                    )
                }

                else -> {
                    // Use the standard CarouselShelf for all other shelves
                    CarouselShelf<Any>(
                        title = shelfType.toTitle(selectedOrg),
                        uiState = shelfState,
                        itemContent = { item ->
                            ShelfItemContent(
                                item = item,
                                discoveryViewModel = discoveryViewModel,
                                navController = navController
                            )
                        },
                        actionContent = {
                            if (shelfType == ShelfType.TRENDING_SONGS) {
                                TextButton(onClick = {
                                    (shelfState as? UiState.Success)?.data?.let {
                                        discoveryViewModel.addAllToQueue(it.filterIsInstan
                                    }
                                }) {
                                    Icon(
                                        Icons.AutoMirrored.Filled.QueueMusic,
                                        null,
                                        modifier = Modifier.size(ButtonDefaults.IconSize)
                                    )
                                    Spacer(Modifier.size(ButtonDefaults.IconSpacing))
                                    Text("Queue")
                                }
```

```
                                } else {
                                    TextButton(onClick = {
                                        navController.navigate(
                                            AppDestinations.fullListViewRoute(
                                                shelfType.toMusicCategoryType(),
                                                selectedOrg
                                            )
                                        )
                                    }) {
                                        Text(stringResource(R.string.action_show_more))
                                    }
                                }
                            }
                        )
                    }
                }
                // --- END OF MODIFICATION ---
            }
        }
    }
}

    @Composable
private fun ShelfItemContent(
    item: Any,
    discoveryViewModel: DiscoveryViewModel,
    navController: NavController
) {
    when (item) {
        is UnifiedDisplayItem -> UnifiedGridItem(item = item, onClick = { discoveryViewModel.p
        is SingingStreamShelfItem -> {
            val displayShell = item.video.toUnifiedDisplayItem(isLiked = false, downloadedSegm
            UnifiedGridItem(
                item = displayShell,
                onClick = {
                    navController.navigate(AppDestinations.videoDetailRoute(item.video.id))
                }
            )
        }
        is PlaylistStub -> PlaylistCard(
            playlist = item,
            onPlaylistClicked = { playlistStub ->
                // Decide what to do based on the type
                if (playlistStub.type.startsWith("radio")) {
                    discoveryViewModel.playRadioPlaylist(playlistStub)
                } else {
                    navController.navigate(AppDestinations.playlistDetailsRoute(playlistStub.i
                }
            }
        )
        is DiscoveryChannel -> ChannelCard(
            channel = item,
            onChannelClicked = { channelId -> navController.navigate("channel_details/$channel
        )
    }
```

```
}


private fun ShelfType.toTitle(orgName: String): String {
    val displayOrg = if (orgName == "All Vtubers") "All" else orgName
    return when (this) {
        ShelfType.RECENT_STREAMS -> "Recent Singing Streams"
        ShelfType.SYSTEM_PLAYLISTS -> "$displayOrg Playlists"
        ShelfType.ARTIST_RADIOS -> "$displayOrg Radios"
        ShelfType.FAN_PLAYLISTS -> "$displayOrg Community Playlists"
        ShelfType.TRENDING_SONGS -> "Trending Songs"
        ShelfType.DISCOVER_CHANNELS -> "Discover $displayOrg"
        ShelfType.FOR_YOU -> "For You"
    }
}

@androidx.annotation.OptIn(UnstableApi::class)
private fun ShelfType.toMusicCategoryType(): MusicCategoryType {
    return when (this) {
        ShelfType.TRENDING_SONGS -> MusicCategoryType.TRENDING
        ShelfType.RECENT_STREAMS -> MusicCategoryType.RECENT_STREAMS
        ShelfType.FAN_PLAYLISTS -> MusicCategoryType.COMMUNITY_PLAYLISTS
        ShelfType.ARTIST_RADIOS -> MusicCategoryType.ARTIST_RADIOS
        // --- START OF MODIFICATION ---
        ShelfType.SYSTEM_PLAYLISTS -> MusicCategoryType.SYSTEM_PLAYLISTS
        ShelfType.DISCOVER_CHANNELS -> MusicCategoryType.DISCOVER_CHANNELS
        // --- END OF MODIFICATION ---
        ShelfType.FOR_YOU -> MusicCategoryType.FAVORITES // For You is a special case of favor
    }
}

// File: java\com\example\holodex\ui\screens\DownloadsScreen.kt
// File: java/com/example/holodex/ui/screens/DownloadsScreen.kt
package com.example.holodex.ui.screens

import androidx.compose.foundation.background
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.PaddingValues
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.aspectRatio
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.grid.GridCells
import androidx.compose.foundation.lazy.grid.LazyVerticalGrid
import androidx.compose.foundation.lazy.grid.items
import androidx.compose.foundation.lazy.items
import androidx.compose.foundation.shape.CircleShape
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.foundation.text.KeyboardActions
```

```
import androidx.compose.foundation.text.KeyboardOptions
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.automirrored.filled.PlaylistPlay
import androidx.compose.material.icons.automirrored.filled.ViewList
import androidx.compose.material.icons.filled.Cancel
import androidx.compose.material.icons.filled.Clear
import androidx.compose.material.icons.filled.Download
import androidx.compose.material.icons.filled.Error
import androidx.compose.material.icons.filled.GridView
import androidx.compose.material.icons.filled.MoreVert
import androidx.compose.material.icons.filled.Pause
import androidx.compose.material.icons.filled.PlayArrow
import androidx.compose.material.icons.filled.Refresh
import androidx.compose.material.icons.filled.Schedule
import androidx.compose.material.icons.filled.Search
import androidx.compose.material.icons.filled.SearchOff
import androidx.compose.material3.ButtonDefaults
import androidx.compose.material3.Card
import androidx.compose.material3.CircularProgressIndicator
import androidx.compose.material3.ExperimentalMaterial3Api
import androidx.compose.material3.Icon
import androidx.compose.material3.IconButton
import androidx.compose.material3.LinearProgressIndicator
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.OutlinedTextField
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Text
import androidx.compose.material3.TextButton
import androidx.compose.material3.TopAppBar
import androidx.compose.material3.TopAppBarDefaults
import androidx.compose.material3.pulltorefresh.PullToRefreshBox
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableIntStateOf
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.rememberCoroutineScope
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.input.nestedscroll.nestedScroll
import androidx.compose.ui.layout.ContentScale
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.platform.LocalFocusManager
import androidx.compose.ui.platform.LocalSoftwareKeyboardController
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.text.input.ImeAction
import androidx.compose.ui.text.style.TextAlign
import androidx.compose.ui.text.style.TextOverflow
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import androidx.hilt.navigation.compose.hiltViewModel
import androidx.lifecycle.compose.collectAsStateWithLifecycle
```

```
import androidx.media3.common.util.UnstableApi
import androidx.navigation.NavController
import coil.compose.AsyncImage
import coil.request.ImageRequest
import com.example.holodex.R
import com.example.holodex.data.db.DownloadStatus
import com.example.holodex.data.db.DownloadedItemEntity
import com.example.holodex.data.db.LikedItemType
import com.example.holodex.playback.domain.model.PlaybackItem
import com.example.holodex.playback.util.formatDurationSecondsToString
import com.example.holodex.ui.AppDestinations
import com.example.holodex.ui.composables.ItemMenuActions
import com.example.holodex.ui.composables.ItemMenuState
import com.example.holodex.ui.composables.ItemOptionsMenu
import com.example.holodex.util.ThumbnailQuality
import com.example.holodex.util.getYouTubeThumbnailUrl
import com.example.holodex.viewmodel.DownloadsViewModel
import com.example.holodex.viewmodel.PlaylistManagementViewModel
import com.example.holodex.viewmodel.VideoListViewModel
import com.example.holodex.viewmodel.toUnifiedDisplayItem
import kotlinx.coroutines.launch

// Data class for clean action handling
data class DownloadItemActions(
    val onPlay: () -> Unit,
    val onDelete: () -> Unit,
    val onRetryDownload: () -> Unit, // For network failures
    val onRetryExport: () -> Unit,   // NEW: For post-processing failures
    val onCancel: () -> Unit,
    val onResume: () -> Unit,
    val playbackItem: PlaybackItem
)

// Helper function to create actions for each download item
@UnstableApi
private fun createDownloadItemActions(
    item: DownloadedItemEntity,
    downloadsViewModel: DownloadsViewModel
): DownloadItemActions {
    return DownloadItemActions(
        onPlay = { downloadsViewModel.playDownloads(item) },
        onDelete = { downloadsViewModel.deleteDownload(item.videoId) },
        onRetryDownload = { downloadsViewModel.retryDownload(item) },
        onRetryExport = { downloadsViewModel.retryExport(item) },
        onCancel = { downloadsViewModel.cancelDownload(item.videoId) },
        onResume = { downloadsViewModel.resumeDownload(item.videoId) },
        playbackItem = downloadsViewModel.mapDownloadToPlaybackItem(item)
    )
}

@UnstableApi
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun DownloadsScreen(
    navController: NavController,
```

```
    downloadsViewModel: DownloadsViewModel = hiltViewModel(),
    playlistManagementViewModel: PlaylistManagementViewModel,
) {
    val filteredDownloads by downloadsViewModel.filteredDownloads.collectAsStateWithLifecycle(
    val searchQuery by downloadsViewModel.searchQuery.collectAsStateWithLifecycle()
    val hasCompletedDownloads =
        filteredDownloads.any { it.downloadStatus == DownloadStatus.COMPLETED }

    var isGridView by remember { mutableStateOf(false) }
    val focusManager = LocalFocusManager.current
    val keyboardController = LocalSoftwareKeyboardController.current
    val scrollBehavior = TopAppBarDefaults.enterAlwaysScrollBehavior()

    var isRefreshing by remember { mutableStateOf(false) }
    val coroutineScope = rememberCoroutineScope()

    Scaffold(
        modifier = Modifier.nestedScroll(scrollBehavior.nestedScrollConnection),
        topBar = {
            TopAppBar(
                title = { Text(stringResource(R.string.bottom_nav_downloads)) },
                actions = {
                    if (hasCompletedDownloads) {
                        TextButton(onClick = { downloadsViewModel.playAllDownloadsShuffled() }
                            Icon(Icons.AutoMirrored.Filled.PlaylistPlay, contentDescription =
                            Spacer(Modifier.size(ButtonDefaults.IconSpacing))
                            Text(stringResource(R.string.action_play_all))
                        }
                    }
                    IconButton(onClick = { isGridView = !isGridView }) {
                        Icon(
                            imageVector = if (isGridView) Icons.AutoMirrored.Filled.ViewList e
                            contentDescription = stringResource(if (isGridView) R.string.actio
                        )
                    }
                },
                scrollBehavior = scrollBehavior
            )
        }
    ) { paddingValues ->
        PullToRefreshBox(
            isRefreshing = isRefreshing,
            onRefresh = {
                coroutineScope.launch {
                    isRefreshing = true
                    try {
                        downloadsViewModel.purgeStaleDownloads()
                    } finally {
                        isRefreshing = false
                    }
                }
            },
            modifier = Modifier
                .padding(paddingValues)
                .fillMaxSize()
```

```
        ) {
            Column(modifier = Modifier.fillMaxSize()) {
                OutlinedTextField(
                    value = searchQuery,
                    onValueChange = { downloadsViewModel.onSearchQueryChanged(it) },
                    modifier = Modifier
                        .fillMaxWidth()
                        .padding(horizontal = 16.dp, vertical = 8.dp),
                    placeholder = { Text(stringResource(R.string.search_your_downloads_hint))
                    leadingIcon = { Icon(Icons.Filled.Search, null) },
                    trailingIcon = {
                        if (searchQuery.isNotEmpty()) {
                            IconButton(onClick = { downloadsViewModel.onSearchQueryChanged("")
                                Icon(Icons.Filled.Clear, stringResource(R.string.action_clear_
                            }
                        }
                    },
                    singleLine = true,
                    keyboardOptions = KeyboardOptions(imeAction = ImeAction.Search),
                    keyboardActions = KeyboardActions(onSearch = {
                        focusManager.clearFocus()
                        keyboardController?.hide()
                    })
                )

                if (filteredDownloads.isEmpty()) {
                    EmptyDownloadsState(isSearching = searchQuery.isNotEmpty())
                } else {
                    if (isGridView) {
                        DownloadsGrid(
                            downloads = filteredDownloads,
                            navController = navController,
                            playlistManagementViewModel = playlistManagementViewModel,
                            downloadsViewModel = downloadsViewModel,
                            createActions = { item ->
                                createDownloadItemActions(item, downloadsViewModel)
                            }
                        )
                    } else {
                        DownloadsList(
                            downloads = filteredDownloads,
                            navController = navController,
                            playlistManagementViewModel = playlistManagementViewModel,
                            downloadsViewModel = downloadsViewModel,
                            createActions = { item ->
                                createDownloadItemActions(item, downloadsViewModel)
                            }
                        )
                    }
                }
            }
        }
    }
}
```

```
@UnstableApi
@Composable
private fun DownloadsList(
    downloads: List<DownloadedItemEntity>,
    navController: NavController,
    playlistManagementViewModel: PlaylistManagementViewModel,
    downloadsViewModel: DownloadsViewModel,
    createActions: (DownloadedItemEntity) -> DownloadItemActions
) {
    LazyColumn(modifier = Modifier.fillMaxSize(), contentPadding = PaddingValues(bottom = 80.d
        items(downloads, key = { it.videoId }) { item ->
            DownloadItemRow(
                item = item,
                actions = createActions(item),
                navController = navController,
                playlistManagementViewModel = playlistManagementViewModel,
                downloadsViewModel = downloadsViewModel
            )
        }
    }
}


@UnstableApi
@Composable
private fun DownloadsGrid(
    downloads: List<DownloadedItemEntity>,
    navController: NavController,
    playlistManagementViewModel: PlaylistManagementViewModel,
    downloadsViewModel: DownloadsViewModel,
    createActions: (DownloadedItemEntity) -> DownloadItemActions
) {
    LazyVerticalGrid(
        columns = GridCells.Adaptive(minSize = 140.dp),
        modifier = Modifier.fillMaxSize(),
        contentPadding = PaddingValues(start = 16.dp, end = 16.dp, top = 8.dp, bottom = 80.dp)
        horizontalArrangement = Arrangement.spacedBy(16.dp),
        verticalArrangement = Arrangement.spacedBy(16.dp)
    ) {
        items(downloads, key = { "grid_${it.videoId}" }) { item ->
            DownloadGridItem(
                item = item,
                actions = createActions(item),
                navController = navController,
                playlistManagementViewModel = playlistManagementViewModel,
                downloadsViewModel = downloadsViewModel
            )
        }
    }
}

@UnstableApi
@OptIn(ExperimentalMaterial3Api::class)
@Composable
private fun DownloadItemRow(
```

```
    item: DownloadedItemEntity,
    navController: NavController,
    actions: DownloadItemActions,
    playlistManagementViewModel: PlaylistManagementViewModel,
    downloadsViewModel: DownloadsViewModel
) {
    var showMenu by remember { mutableStateOf(false) }

    val videoListViewModel: VideoListViewModel = hiltViewModel()

    val videoId = remember(item.videoId) { item.videoId.split('_').first() }
    val thumbnailUrls = remember(videoId, item.artworkUrl) {
        listOfNotNull(item.artworkUrl) + getYouTubeThumbnailUrl(videoId, ThumbnailQuality.MEDI
    }
    var currentUrlIndex by remember(thumbnailUrls) { mutableIntStateOf(0) }

    Card(
        modifier = Modifier
            .fillMaxWidth()
            .padding(horizontal = 16.dp, vertical = 6.dp),
        onClick = { if (item.downloadStatus == DownloadStatus.COMPLETED) actions.onPlay() },
        enabled = item.downloadStatus == DownloadStatus.COMPLETED
    ) {
        Column {
            if (item.downloadStatus == DownloadStatus.DOWNLOADING && item.progress > 0) {
                LinearProgressIndicator(
                    progress = { item.progress / 100f },
                    modifier = Modifier.fillMaxWidth(),
                )
            }

            Row(
                modifier = Modifier.padding(12.dp),
                verticalAlignment = Alignment.CenterVertically
            ) {
                AsyncImage(
                    model = ImageRequest.Builder(LocalContext.current)
                        .data(thumbnailUrls.getOrNull(currentUrlIndex))
                        .placeholder(R.drawable.ic_placeholder_image)
                        .error(R.drawable.ic_error_image)
                        .crossfade(true).build(),
                    onError = {
                        if (currentUrlIndex < thumbnailUrls.lastIndex) currentUrlIndex++
                    },
                    contentDescription = "Artwork for ${item.title}",
                    contentScale = ContentScale.Crop,
                    modifier = Modifier
                        .size(56.dp)
                        .clip(RoundedCornerShape(8.dp))
                )

                Column(
                    modifier = Modifier
                        .weight(1f)
                        .padding(horizontal = 12.dp),
```

```
        verticalArrangement = Arrangement.Center
    ) {
        Text(
            item.title,
            maxLines = 1,
            overflow = TextOverflow.Ellipsis,
            style = MaterialTheme.typography.titleSmall
        )
        Text(
            text = item.artistText,
            style = MaterialTheme.typography.bodyMedium,
            color = MaterialTheme.colorScheme.onSurfaceVariant,
            maxLines = 1,
            overflow = TextOverflow.Ellipsis
        )
        Row(
            horizontalArrangement = Arrangement.spacedBy(8.dp),
            verticalAlignment = Alignment.CenterVertically
        ) {
            Text(
                text = formatDurationSecondsToString(item.durationSec),
                style = MaterialTheme.typography.bodySmall,
                color = MaterialTheme.colorScheme.onSurfaceVariant
            )
            if (item.downloadStatus == DownloadStatus.DOWNLOADING && item.progress
                Text(
                    text = "${item.progress}%",
                    style = MaterialTheme.typography.bodySmall,
                    color = MaterialTheme.colorScheme.primary
                )
            }
        }
    }

    DownloadStatusIndicator(
        status = item.downloadStatus,
        progress = item.progress,
        onCancel = actions.onCancel,
        onResume = actions.onResume,
        onRetryDownload = actions.onRetryDownload,
        onRetryExport = actions.onRetryExport,
        onDelete = actions.onDelete,
        onShowMenu = { showMenu = true }
    )

    Box {
        val menuState = remember(item) {
            ItemMenuState(
                isDownloaded = true, isSegment = true, canBeDownloaded = false,
                shareUrl = "https://music.holodex.net/watch/${item.videoId.substri
                videoId = item.videoId.substringBeforeLast('_'), channelId = item.
            )
        }

        val menuActions = remember(item) {
```

```
                        ItemMenuActions(
                            onAddToQueue = { videoListViewModel.addVideoOrItsSegmentsToQueue(a
                            onAddToPlaylist = {
                                val unifiedItem = downloadsViewModel.mapDownloadToPlaybackItem
                                playlistManagementViewModel.prepareItemForPlaylistAddition(uni
                            },
                            onShare = { /* Handled internally by the menu */ },
                            onDownload = { /* No-op, already downloaded */ },
                            onDelete = actions.onDelete,
                            onGoToVideo = { videoId -> navController.navigate(AppDestinations.
                            onGoToArtist = { channelId -> navController.navigate("channel_deta
                        )
                    }

                    if (item.downloadStatus == DownloadStatus.COMPLETED) {
                        ItemOptionsMenu(
                            state = menuState,
                            actions = menuActions,
                            expanded = showMenu,
                            onDismissRequest = { showMenu = false }
                        )
                    }
                }
            }
        }
    }
}

@UnstableApi
@Composable
private fun DownloadGridItem(
    item: DownloadedItemEntity,
    navController: NavController,
    actions: DownloadItemActions,
    playlistManagementViewModel: PlaylistManagementViewModel,
    downloadsViewModel: DownloadsViewModel
) {
    var showMenu by remember { mutableStateOf(false) }
    val videoListViewModel: VideoListViewModel = hiltViewModel()

    val videoId = remember(item.videoId) { item.videoId.split('_').first() }
    val thumbnailUrls = remember(videoId, item.artworkUrl) {
        listOfNotNull(item.artworkUrl) + getYouTubeThumbnailUrl(videoId, ThumbnailQuality.HIGH
    }
    var currentUrlIndex by remember(thumbnailUrls) { mutableIntStateOf(0) }

    Card(
        onClick = { if (item.downloadStatus == DownloadStatus.COMPLETED) actions.onPlay() },
        enabled = item.downloadStatus == DownloadStatus.COMPLETED
    ) {
        Column {
            Box(contentAlignment = Alignment.Center) {
                AsyncImage(
                    model = ImageRequest.Builder(LocalContext.current)
                        .data(thumbnailUrls.getOrNull(currentUrlIndex))
```

```
            .placeholder(R.drawable.ic_placeholder_image)
            .error(R.drawable.ic_error_image)
            .crossfade(true).build(),
        onError = { if (currentUrlIndex < thumbnailUrls.lastIndex) currentUrlIndex
        contentDescription = "Artwork for ${item.title}",
        contentScale = ContentScale.Crop,
        modifier = Modifier
            .fillMaxWidth()
            .aspectRatio(1f)
    )

    if (item.downloadStatus != DownloadStatus.COMPLETED) {
        Box(
            modifier = Modifier
                .matchParentSize()
                .background(Color.Black.copy(alpha = 0.6f)),
            contentAlignment = Alignment.Center
        ) {
            when (item.downloadStatus) {
                DownloadStatus.DOWNLOADING -> {
                    Column(horizontalAlignment = Alignment.CenterHorizontally, ver
                        CircularProgressIndicator(progress = { item.progress / 100
                        if (item.progress > 0) Text(text = "${item.progress}%", st
                        IconButton(onClick = actions.onCancel, modifier = Modifier
                    }
                }
                DownloadStatus.ENQUEUED -> {
                    Column(horizontalAlignment = Alignment.CenterHorizontally, ver
                        Icon(Icons.Filled.Schedule, contentDescription = "Queued",
                        Text(text = stringResource(R.string.status_queued), style
                        IconButton(onClick = actions.onCancel, modifier = Modifier
                    }
                }
                DownloadStatus.PAUSED -> {
                    Column(horizontalAlignment = Alignment.CenterHorizontally, ver
                        Icon(Icons.Filled.Pause, contentDescription = "Paused", ti
                        Text(text = stringResource(R.string.status_paused), style
                        Row(horizontalArrangement = Arrangement.spacedBy(8.dp)) {
                            IconButton(onClick = actions.onResume, modifier = Modi
                            IconButton(onClick = actions.onCancel, modifier = Modi
                        }
                    }
                }
                DownloadStatus.FAILED, DownloadStatus.EXPORT_FAILED -> {
                    Column(horizontalAlignment = Alignment.CenterHorizontally, ver
                        Icon(Icons.Filled.Error, contentDescription = "Failed", ti
                        Text(text = stringResource(R.string.status_failed), style
                        Row(horizontalArrangement = Arrangement.spacedBy(8.dp)) {
                            IconButton(onClick = if (item.downloadStatus == Downlo
                            IconButton(onClick = actions.onDelete, modifier = Modi
                        }
                    }
                }
                else -> Icon(Icons.Filled.Download, contentDescription = "Download
            }
```

```
            }
        }

        if (item.downloadStatus == DownloadStatus.COMPLETED) {
            Box(
                modifier = Modifier.matchParentSize(),
                contentAlignment = Alignment.TopEnd
            ) {
                IconButton(
                    onClick = { showMenu = true },
                    modifier = Modifier.padding(4.dp)
                ) {
                    Icon(
                        imageVector = Icons.Default.MoreVert,
                        contentDescription = stringResource(R.string.action_more_optio
                        tint = Color.White,
                        modifier = Modifier
                            .background(Color.Black.copy(alpha = 0.5f), CircleShape)
                            .padding(4.dp)
                    )
                }
            }
        }
    }

    Column(modifier = Modifier.padding(8.dp)) {
        Text(
            text = item.title,
            style = MaterialTheme.typography.titleSmall,
            maxLines = 2,
            overflow = TextOverflow.Ellipsis
        )
        Text(
            text = item.artistText,
            style = MaterialTheme.typography.bodySmall,
            maxLines = 1,
            overflow = TextOverflow.Ellipsis,
            color = MaterialTheme.colorScheme.onSurfaceVariant
        )
    }
}

Box {
    val menuState = remember(item) {
        ItemMenuState(
            isDownloaded = true, isSegment = true, canBeDownloaded = false,
            shareUrl = "https://music.holodex.net/watch/${item.videoId.substringBefore
            videoId = item.videoId.substringBeforeLast('_'), channelId = item.channelI
        )
    }

    val menuActions = remember(item) {
        ItemMenuActions(
            onAddToQueue = { videoListViewModel.addVideoOrItsSegmentsToQueue(actions.p
            onAddToPlaylist = {
```

```
                        val unifiedItem = downloadsViewModel.mapDownloadToPlaybackItem(item).t
                        playlistManagementViewModel.prepareItemForPlaylistAddition(unifiedItem
                    },
                    onShare = { /* Handled internally */ },
                    onDownload = { /* No-op */ },
                    onDelete = actions.onDelete,
                    onGoToVideo = { videoId -> navController.navigate(AppDestinations.videoDet
                    onGoToArtist = { channelId -> navController.navigate("channel_details/$cha
                )
            }

            if (item.downloadStatus == DownloadStatus.COMPLETED) {
                ItemOptionsMenu(
                    state = menuState,
                    actions = menuActions,
                    expanded = showMenu,
                    onDismissRequest = { showMenu = false }
                )
            }
        }
    }
}


@Composable
private fun DownloadStatusIndicator(
    status: DownloadStatus,
    progress: Int,
    onCancel: () -> Unit,
    onResume: () -> Unit,
    onRetryDownload: () -> Unit,
    onRetryExport: () -> Unit,
    onDelete: () -> Unit,
    onShowMenu: () -> Unit
) {
    Box {
        when (status) {
            DownloadStatus.COMPLETED -> {
                IconButton(onClick = onShowMenu) {
                    Icon(Icons.Default.MoreVert, contentDescription = stringResource(R.string.
                }
            }
            DownloadStatus.DOWNLOADING -> {
                Row(verticalAlignment = Alignment.CenterVertically, horizontalArrangement = Ar
                    if (progress > 0) {
                        Box(contentAlignment = Alignment.Center) {
                            CircularProgressIndicator(progress = { progress / 100f }, modifier
                            Text(text = "$progress", style = MaterialTheme.typography.labelSma
                        }
                    } else {
                        CircularProgressIndicator(modifier = Modifier.size(24.dp), strokeWidth
                    }
                    IconButton(onClick = onCancel) {
                        Icon(Icons.Filled.Cancel, contentDescription = stringResource(R.string
                    }
```

```
                }
            }
            DownloadStatus.ENQUEUED -> {
                Row(verticalAlignment = Alignment.CenterVertically, horizontalArrangement = Ar
                    Icon(Icons.Filled.Schedule, contentDescription = stringResource(R.string.s
                    IconButton(onClick = onCancel) {
                        Icon(Icons.Filled.Cancel, contentDescription = stringResource(R.string
                    }
                }
            }
            DownloadStatus.PAUSED -> {
                Row(verticalAlignment = Alignment.CenterVertically, horizontalArrangement = Ar
                    IconButton(onClick = onResume) {
                        Icon(Icons.Filled.PlayArrow, contentDescription = stringResource(R.str
                    }
                    IconButton(onClick = onCancel) {
                        Icon(Icons.Filled.Cancel, contentDescription = stringResource(R.string
                    }
                }
            }
            DownloadStatus.FAILED -> {
                Row(verticalAlignment = Alignment.CenterVertically, horizontalArrangement = Ar
                    IconButton(onClick = onRetryDownload) {
                        Icon(Icons.Filled.Refresh, contentDescription = stringResource(R.strin
                    }
                    IconButton(onClick = onDelete) {
                        Icon(Icons.Filled.Clear, contentDescription = stringResource(R.string.
                    }
                }
            }
            DownloadStatus.EXPORT_FAILED -> {
                Row(verticalAlignment = Alignment.CenterVertically, horizontalArrangement = Ar
                    IconButton(onClick = onRetryExport) {
                        Icon(Icons.Filled.Refresh, contentDescription = stringResource(R.strin
                    }
                    IconButton(onClick = onDelete) {
                        Icon(Icons.Filled.Clear, contentDescription = stringResource(R.string.
                    }
                }
            }
            DownloadStatus.NOT_DOWNLOADED, DownloadStatus.DELETING -> {
                CircularProgressIndicator(modifier = Modifier.size(24.dp), strokeWidth = 2.dp)
            }
            DownloadStatus.PROCESSING -> Spacer(modifier = Modifier.size(48.dp))
        }
    }
}

@Composable
private fun EmptyDownloadsState(isSearching: Boolean) {
    Box(
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp),
        contentAlignment = Alignment.Center
```

```
    ) {
        Column(
            horizontalAlignment = Alignment.CenterHorizontally,
            verticalArrangement = Arrangement.spacedBy(16.dp),
            modifier = Modifier.padding(bottom = 50.dp)
        ) {
            Icon(
                imageVector = if (isSearching) Icons.Filled.SearchOff else Icons.Filled.Downlo
                contentDescription = null,
                modifier = Modifier.size(64.dp),
                tint = MaterialTheme.colorScheme.onSurfaceVariant.copy(alpha = 0.6f)
            )
            Text(
                text = if (isSearching) stringResource(R.string.message_no_search_results_down
                style = MaterialTheme.typography.titleMedium,
                color = MaterialTheme.colorScheme.onSurfaceVariant,
                textAlign = TextAlign.Center
            )
        }
    }
}

// File: java\com\example\holodex\ui\screens\EditablePlaylistHeader.kt
package com.example.holodex.ui.screens

import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.OutlinedTextField
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Modifier
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.unit.dp
import com.example.holodex.R
import com.example.holodex.data.db.PlaylistEntity
import kotlinx.coroutines.FlowPreview
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.debounce

@OptIn(FlowPreview::class)
@Composable
fun EditablePlaylistHeader(
    playlist: PlaylistEntity,
    onNameChange: (String) -> Unit,
    onDescriptionChange: (String) -> Unit,
    modifier: Modifier = Modifier
```

```
) {
    var name by remember(playlist.name) { mutableStateOf(playlist.name ?: "") }
    var description by remember(playlist.description) { mutableStateOf(playlist.description ?:

    // Use StateFlows with debounce to avoid excessive recompositions on every keystroke
    val nameFlow = remember { MutableStateFlow(name) }
    val descriptionFlow = remember { MutableStateFlow(description) }

    LaunchedEffect(Unit) {
        nameFlow.debounce(300).collect { onNameChange(it) }
    }
    LaunchedEffect(Unit) {
        descriptionFlow.debounce(300).collect { onDescriptionChange(it) }
    }

    Column(
        modifier = modifier
            .fillMaxWidth()
            .padding(16.dp),
        verticalArrangement = Arrangement.spacedBy(8.dp)
    ) {
        OutlinedTextField(
            value = name,
            onValueChange = {
                name = it
                nameFlow.value = it
            },
            label = { Text(stringResource(R.string.hint_playlist_name)) },
            modifier = Modifier.fillMaxWidth(),
            textStyle = MaterialTheme.typography.headlineSmall.copy(fontWeight = FontWeight.Bo
            singleLine = true
        )
        OutlinedTextField(
            value = description,
            onValueChange = {
                description = it
                descriptionFlow.value = it
            },
            label = { Text(stringResource(R.string.hint_playlist_description_optional)) },
            modifier = Modifier.fillMaxWidth(),
            textStyle = MaterialTheme.typography.bodyMedium,
            maxLines = 3
        )
    }
}

// File: java\com\example\holodex\ui\screens\ExternalChannelScreen.kt
// File: java/com/example/holodex/ui/screens/ExternalChannelScreen.kt
package com.example.holodex.ui.screens

import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.PaddingValues
import androidx.compose.foundation.layout.fillMaxSize
```

```
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.foundation.lazy.rememberLazyListState
import androidx.compose.foundation.shape.CircleShape
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.automirrored.filled.ArrowBack
import androidx.compose.material3.CircularProgressIndicator
import androidx.compose.material3.ExperimentalMaterial3Api
import androidx.compose.material3.Icon
import androidx.compose.material3.IconButton
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Text
import androidx.compose.material3.TopAppBar
import androidx.compose.material3.TopAppBarDefaults
import androidx.compose.runtime.Composable
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.runtime.derivedStateOf
import androidx.compose.runtime.getValue
import androidx.compose.runtime.remember
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.layout.ContentScale
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.text.style.TextAlign
import androidx.compose.ui.unit.dp
import androidx.hilt.navigation.compose.hiltViewModel
import androidx.lifecycle.compose.collectAsStateWithLifecycle
import androidx.media3.common.util.UnstableApi
import androidx.navigation.NavController
import coil.compose.AsyncImage
import com.example.holodex.R
import com.example.holodex.ui.composables.EmptyState
import com.example.holodex.ui.composables.ErrorStateWithRetry
import com.example.holodex.ui.composables.LoadingState
import com.example.holodex.ui.composables.SimpleProcessedBackground
import com.example.holodex.ui.composables.UnifiedListItem
import com.example.holodex.util.findActivity
import com.example.holodex.viewmodel.ExternalChannelViewModel
import com.example.holodex.viewmodel.FavoritesViewModel
import com.example.holodex.viewmodel.PlaylistManagementViewModel
import com.example.holodex.viewmodel.VideoListViewModel
import com.example.holodex.viewmodel.mappers.toPlaybackItem
import com.example.holodex.viewmodel.state.UiState

@OptIn(ExperimentalMaterial3Api::class, UnstableApi::class)
@Composable
fun ExternalChannelScreen(
    navController: NavController,
```

```
    onNavigateUp: () -> Unit,
    channelViewModel: ExternalChannelViewModel = hiltViewModel(),
    favoritesViewModel: FavoritesViewModel = hiltViewModel(),
    videoListViewModel: VideoListViewModel = hiltViewModel(findActivity()),
    playlistManagementViewModel: PlaylistManagementViewModel = hiltViewModel(findActivity())
) {
    val details by channelViewModel.channelDetails.collectAsStateWithLifecycle()
    val musicItems by channelViewModel.musicItems.collectAsStateWithLifecycle()
    val uiState by channelViewModel.uiState.collectAsStateWithLifecycle()
    val dynamicTheme by channelViewModel.dynamicTheme.collectAsStateWithLifecycle()
    val listState = rememberLazyListState()

    // Collect the new pagination states
    val isLoadingMore by channelViewModel.isLoadingMore.collectAsStateWithLifecycle()
    val endOfList by channelViewModel.endOfList.collectAsStateWithLifecycle()

    val shouldLoadMore by remember {
        derivedStateOf {
            val layoutInfo = listState.layoutInfo
            if (layoutInfo.visibleItemsInfo.isEmpty()) return@derivedStateOf false
            val lastVisibleItem = layoutInfo.visibleItemsInfo.last()
            lastVisibleItem.index >= layoutInfo.totalItemsCount - 5
        }
    }

    LaunchedEffect(shouldLoadMore) {
        if (shouldLoadMore) {
            channelViewModel.loadMoreMusic()
        }
    }

    Scaffold(
        topBar = {
            TopAppBar(
                title = { details?.name?.let { Text(it) } },
                navigationIcon = { IconButton(onClick = onNavigateUp) { Icon(Icons.AutoMirrore
                colors = TopAppBarDefaults.topAppBarColors(containerColor = Color.Transparent,
            )
        },
        containerColor = Color.Transparent
    ) { paddingValues ->
        Box(modifier = Modifier.fillMaxSize()) {
            SimpleProcessedBackground(artworkUri = details?.photoUrl, dynamicColor = dynamicTh

            when (val state = uiState) {
                is UiState.Loading -> LoadingState(message = "Loading music...")
                is UiState.Error -> ErrorStateWithRetry(message = state.message, onRetry = { c
                is UiState.Success -> {
                    if (musicItems.isEmpty()) {
                        EmptyState(message = "No music content found for this channel.", onRef
                    } else {
                        LazyColumn(
                            state = listState,
                            modifier = Modifier.padding(paddingValues).fillMaxSize(),
                            contentPadding = PaddingValues(bottom = 80.dp)
```

```
                        ) {
                            item {
                                details?.let { ExternalChannelHeader(details = it) }
                            }
                            items(musicItems, key = { it.stableId }) { item ->
                                UnifiedListItem(
                                    item = item,
                                    onItemClicked = { videoListViewModel.playFavoriteOrLikedSe
                                    navController = navController,
                                    videoListViewModel = videoListViewModel,
                                    favoritesViewModel = favoritesViewModel,
                                    playlistManagementViewModel = playlistManagementViewModel,
                                    isExternal = true
                                )
                            }

                            // *** ADD THE PAGINATION FOOTER UI ***
                            item {
                                if (isLoadingMore) {
                                    Box(
                                        modifier = Modifier.fillMaxWidth().padding(vertical =
                                        contentAlignment = Alignment.Center
                                    ) {
                                        CircularProgressIndicator(modifier = Modifier.size(36.
                                    }
                                } else if (endOfList) {
                                    Text(
                                        text = stringResource(R.string.message_youve_reached_t
                                        modifier = Modifier.fillMaxWidth().padding(16.dp),
                                        textAlign = TextAlign.Center,
                                        style = MaterialTheme.typography.bodySmall,
                                        color = Color.White.copy(alpha = 0.7f)
                                    )
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}

@Composable
private fun ExternalChannelHeader(details: com.example.holodex.data.model.discovery.ChannelDet
    Column(
        modifier = Modifier
            .fillMaxWidth()
            .padding(16.dp),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.spacedBy(12.dp)
    ) {
        AsyncImage(
            model = details.photoUrl,
            contentDescription = "Channel Avatar",
```

```
            modifier = Modifier.size(96.dp).clip(CircleShape),
            contentScale = ContentScale.Crop
        )
        Text(details.name, style = MaterialTheme.typography.headlineMedium, fontWeight = FontW
    }
}

// File: java\com\example\holodex\ui\screens\FavoritesScreen.kt
// File: java/com/example/holodex/ui/screens/FavoritesScreen.kt
package com.example.holodex.ui.screens

import android.widget.Toast
import androidx.compose.animation.core.animateFloatAsState
import androidx.compose.foundation.clickable
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.PaddingValues
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.aspectRatio
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.heightIn
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.layout.width
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.LazyRow
import androidx.compose.foundation.lazy.grid.GridCells
import androidx.compose.foundation.lazy.grid.LazyVerticalGrid
import androidx.compose.foundation.lazy.grid.items
import androidx.compose.foundation.lazy.items
import androidx.compose.foundation.shape.CircleShape
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.ExpandMore
import androidx.compose.material3.Card
import androidx.compose.material3.HorizontalDivider
import androidx.compose.material3.Icon
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.draw.rotate
import androidx.compose.ui.layout.ContentScale
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.text.style.TextAlign
import androidx.compose.ui.text.style.TextOverflow
```

```
import androidx.compose.ui.unit.dp
import androidx.hilt.navigation.compose.hiltViewModel
import androidx.media3.common.util.UnstableApi
import androidx.navigation.NavController
import coil.compose.AsyncImage
import coil.request.ImageRequest
import com.example.holodex.R
import com.example.holodex.data.db.ExternalChannelEntity
import com.example.holodex.data.db.FavoriteChannelEntity
import com.example.holodex.ui.AppDestinations
import com.example.holodex.ui.composables.LoadingSkeleton
import com.example.holodex.ui.composables.UnifiedListItem
import com.example.holodex.viewmodel.FavoritesSideEffect
import com.example.holodex.viewmodel.FavoritesViewModel
import com.example.holodex.viewmodel.PlaylistManagementViewModel
import com.example.holodex.viewmodel.UnifiedDisplayItem
import com.example.holodex.viewmodel.VideoListViewModel
import com.example.holodex.viewmodel.mappers.toPlaybackItem
import org.orbitmvi.orbit.compose.collectAsState
import org.orbitmvi.orbit.compose.collectSideEffect

@OptIn(UnstableApi::class)
@Composable
fun FavoritesScreen(
    modifier: Modifier = Modifier,
    isGridView: Boolean,
    videoListViewModel: VideoListViewModel,
    playlistManagementViewModel: PlaylistManagementViewModel,
    navController: NavController,
    favoritesViewModel: FavoritesViewModel = hiltViewModel()
) {
    // *** THE FIX: Collect the single state object from the Orbit container ***
    val state by favoritesViewModel.collectAsState()
    val context = LocalContext.current

    // Handle one-time side effects like toasts
    favoritesViewModel.collectSideEffect { sideEffect ->
        when (sideEffect) {
            is FavoritesSideEffect.ShowToast -> {
                Toast.makeText(context, sideEffect.message, Toast.LENGTH_SHORT).show()
            }
        }
    }

    var isChannelsExpanded by remember { mutableStateOf(true) }
    var isFavoritesExpanded by remember { mutableStateOf(true) }
    var isSegmentsExpanded by remember { mutableStateOf(true) }

    // Use the isLoading flag from the state
    if (state.isLoading) {
        LoadingSkeleton(itemCount = 8, modifier = modifier.padding(16.dp))
        return
    }

    LazyColumn(modifier = modifier.fillMaxSize(), contentPadding = PaddingValues(bottom = 80.d
```

```
if (state.favoriteChannels.isNotEmpty()) {
    item {
        ExpandableSectionHeader(
            title = stringResource(R.string.category_favorite_channels),
            itemCount = state.favoriteChannels.size,
            isExpanded = isChannelsExpanded,
            onToggle = { isChannelsExpanded = !isChannelsExpanded }
        )
    }
    if (isChannelsExpanded) {
        item {
            FavoriteChannelsRow(
                channels = state.favoriteChannels,
                onChannelClick = { channel ->
                    val (id, isExternal) = when (channel) {
                        is FavoriteChannelEntity -> channel.id to false
                        is ExternalChannelEntity -> channel.channelId to true
                        else -> null to false
                    }
                    if (id != null) {
                        val route =
                            if (isExternal) "external_channel_details/$id" else "chann
                        navController.navigate(route)
                    }
                }
            )
        }
    }
    item { HorizontalDivider(modifier = Modifier.padding(vertical = 8.dp)) }
}

item {
    ExpandableSectionHeader(
        title = stringResource(R.string.category_favorites),
        itemCount = state.unifiedFavoritedVideos.size,
        isExpanded = isFavoritesExpanded,
        onToggle = { isFavoritesExpanded = !isFavoritesExpanded }
    )
}

if (isFavoritesExpanded && state.unifiedFavoritedVideos.isNotEmpty()) {
    if (isGridView) {
        item {
            FavoritesGrid(
                items = state.unifiedFavoritedVideos,
                onItemClicked = { item ->
                    navController.navigate(
                        AppDestinations.videoDetailRoute(
                            item.videoId
                        )
                    )
                }
            )
        }
    } else {
```

```
            items(items = state.unifiedFavoritedVideos, key = { it.stableId }) { item ->
                UnifiedListItem(
                    item = item,
                    onItemClicked = {
                        navController.navigate(
                            AppDestinations.videoDetailRoute(
                                item.videoId
                            )
                        )
                    },
                    videoListViewModel = videoListViewModel,
                    favoritesViewModel = favoritesViewModel,
                    playlistManagementViewModel = playlistManagementViewModel,
                    navController = navController
                )
            }
        }
    }

    item { HorizontalDivider(modifier = Modifier.padding(vertical = 8.dp)) }

    item {
        ExpandableSectionHeader(
            title = stringResource(R.string.category_liked_segments),
            itemCount = state.unifiedLikedSegments.size,
            isExpanded = isSegmentsExpanded,
            onToggle = { isSegmentsExpanded = !isSegmentsExpanded }
        )
    }

    if (isSegmentsExpanded && state.unifiedLikedSegments.isNotEmpty()) {
        if (isGridView) {
            item {
                FavoritesGrid(
                    items = state.unifiedLikedSegments,
                    onItemClicked = { item ->
                        videoListViewModel.playFavoriteOrLikedSegmentItem(
                            item.toPlaybackItem()
                        )
                    }
                )
            }
        } else {
            items(items = state.unifiedLikedSegments, key = { it.stableId }) { item ->
                UnifiedListItem(
                    item = item,
                    onItemClicked = { videoListViewModel.playFavoriteOrLikedSegmentItem(it
                    videoListViewModel = videoListViewModel,
                    navController = navController,
                    playlistManagementViewModel = playlistManagementViewModel,
                    favoritesViewModel = favoritesViewModel
                )
            }
        }
    }
}
```

```
    }
}

// Rest of the file (helper composables) remains the same

@Composable
private fun UnifiedGridItem(
    item: UnifiedDisplayItem,
    onClick: () -> Unit,
    modifier: Modifier = Modifier
) {
    Card(onClick = onClick, modifier = modifier) {
        Column {
            AsyncImage(
                model = ImageRequest.Builder(LocalContext.current)
                    .data(item.artworkUrls.firstOrNull())
                    .placeholder(R.drawable.ic_placeholder_image)
                    .error(R.drawable.ic_error_image)
                    .crossfade(true)
                    .build(),
                contentDescription = "Artwork for ${item.title}",
                contentScale = ContentScale.Crop,
                modifier = Modifier
                    .fillMaxWidth()
                    .aspectRatio(1f)
            )
            Column(Modifier.padding(8.dp)) {
                Text(
                    text = item.title,
                    style = MaterialTheme.typography.titleSmall,
                    maxLines = 2,
                    overflow = TextOverflow.Ellipsis
                )
                Text(
                    text = item.artistText,
                    style = MaterialTheme.typography.bodySmall,
                    maxLines = 1,
                    overflow = TextOverflow.Ellipsis,
                    color = MaterialTheme.colorScheme.onSurfaceVariant
                )
            }
        }
    }
}

@Composable
private fun FavoritesGrid(
    items: List<UnifiedDisplayItem>,
    onItemClicked: (UnifiedDisplayItem) -> Unit,
) {
    LazyVerticalGrid(
        columns = GridCells.Adaptive(minSize = 128.dp),
        contentPadding = PaddingValues(16.dp),
        horizontalArrangement = Arrangement.spacedBy(16.dp),
        verticalArrangement = Arrangement.spacedBy(16.dp),
```

```
        modifier = Modifier.heightIn(min = 1.dp), // Prevents crash on empty list
        userScrollEnabled = false
    ) {
        items(items, key = { it.stableId }) { item ->
            UnifiedGridItem(item = item, onClick = { onItemClicked(item) })
        }
    }
}

@Composable
private fun ExpandableSectionHeader(
    title: String,
    itemCount: Int,
    isExpanded: Boolean,
    onToggle: () -> Unit
) {
    val rotationAngle by animateFloatAsState(
        targetValue = if (isExpanded) 0f else -90f,
        label = "expansion_arrow"
    )
    Row(
        modifier = Modifier
            .fillMaxWidth()
            .clickable { onToggle() }
            .padding(horizontal = 16.dp, vertical = 12.dp),
        verticalAlignment = Alignment.CenterVertically
    ) {
        Text(
            text = "$title ($itemCount)",
            style = MaterialTheme.typography.headlineSmall,
            modifier = Modifier.weight(1f)
        )
        Icon(
            imageVector = Icons.Default.ExpandMore,
            contentDescription = if (isExpanded) "Collapse" else "Expand",
            modifier = Modifier.rotate(rotationAngle)
        )
    }
}

@Composable
private fun FavoriteChannelsRow(channels: List<Any>, onChannelClick: (Any) -> Unit) {
    LazyRow(
        contentPadding = PaddingValues(horizontal = 16.dp, vertical = 8.dp),
        horizontalArrangement = Arrangement.spacedBy(16.dp)
    ) {
        items(channels) { channel ->
            val (photoUrl, name, id) = when (channel) {
                is FavoriteChannelEntity -> Triple(channel.photoUrl, channel.name, channel.id)
                is ExternalChannelEntity -> Triple(
                    channel.photoUrl,
                    channel.name,
                    channel.channelId
                )
```

```
                else -> Triple(null, "Unknown", null)
            }
            Column(
                horizontalAlignment = Alignment.CenterHorizontally,
                modifier = Modifier
                    .width(80.dp)
                    .clickable(enabled = id != null) { onChannelClick(channel) }
            ) {
                AsyncImage(
                    model = photoUrl,
                    contentDescription = name,
                    modifier = Modifier
                        .size(64.dp)
                        .clip(CircleShape),
                    contentScale = ContentScale.Crop
                )
                Spacer(Modifier.height(4.dp))
                Text(
                    text = name ?: stringResource(R.string.unknown_channel),
                    style = MaterialTheme.typography.bodySmall,
                    maxLines = 2,
                    overflow = TextOverflow.Ellipsis,
                    textAlign = TextAlign.Center
                )
            }
        }
    }
}

// File: java\com\example\holodex\ui\screens\ForYouScreen.kt
// File: java/com/example/holodex/ui/screens/ForYouScreen.kt

package com.example.holodex.ui.screens

import android.widget.Toast
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.PaddingValues
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.automirrored.filled.ArrowBack
import androidx.compose.material3.ExperimentalMaterial3Api
import androidx.compose.material3.Icon
import androidx.compose.material3.IconButton
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Text
import androidx.compose.material3.TextButton
import androidx.compose.material3.TopAppBar
import androidx.compose.runtime.Composable
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.runtime.getValue
import androidx.compose.runtime.remember
import androidx.compose.ui.Modifier
```

```
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.unit.dp
import androidx.hilt.navigation.compose.hiltViewModel
import androidx.lifecycle.compose.collectAsStateWithLifecycle
import androidx.media3.common.util.UnstableApi
import androidx.navigation.NavController
import com.example.holodex.R
import com.example.holodex.data.model.discovery.DiscoveryChannel
import com.example.holodex.data.model.discovery.PlaylistStub
import com.example.holodex.data.model.discovery.SingingStreamShelfItem
import com.example.holodex.ui.AppDestinations
import com.example.holodex.ui.composables.CarouselShelf
import com.example.holodex.ui.composables.ChannelCard
import com.example.holodex.ui.composables.ErrorStateWithRetry
import com.example.holodex.ui.composables.LoadingState
import com.example.holodex.ui.composables.PlaylistCard
import com.example.holodex.ui.composables.UnifiedGridItem
import com.example.holodex.viewmodel.DiscoveryViewModel
import com.example.holodex.viewmodel.VideoListViewModel
import com.example.holodex.viewmodel.mappers.toUnifiedDisplayItem
import com.example.holodex.viewmodel.state.UiState
import kotlinx.coroutines.flow.collectLatest

@androidx.annotation.OptIn(UnstableApi::class)
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun ForYouScreen(
    navController: NavController
) {
    val discoveryViewModel: DiscoveryViewModel = hiltViewModel()
    val videoListViewModel: VideoListViewModel = hiltViewModel()

    val forYouState by discoveryViewModel.forYouState.collectAsStateWithLifecycle()
    val context = LocalContext.current

    LaunchedEffect(Unit) {
        discoveryViewModel.loadForYouContent()
        discoveryViewModel.transientMessage.collectLatest { message ->
            Toast.makeText(context, message, Toast.LENGTH_SHORT).show()
        }
    }

    Scaffold(
        topBar = {
            TopAppBar(
                title = { Text(stringResource(R.string.shelf_title_for_you)) },
                navigationIcon = {
                    IconButton(onClick = { navController.popBackStack() }) {
                        Icon(Icons.AutoMirrored.Filled.ArrowBack, "Back")
                    }
                }
            )
        }
    ) { paddingValues ->
```

```
Box(modifier = Modifier
    .padding(paddingValues)
    .fillMaxSize()) {
    when (val state = forYouState) {
        is UiState.Loading -> LoadingState(message = "Loading your personalized conten
        is UiState.Error -> ErrorStateWithRetry(
            message = state.message,
            onRetry = { discoveryViewModel.loadForYouContent() })

        is UiState.Success -> {
            val data = state.data
            LazyColumn(
                modifier = Modifier.fillMaxSize(),
                contentPadding = PaddingValues(vertical = 16.dp),
                verticalArrangement = Arrangement.spacedBy(24.dp)
            ) {
                item {
                    val uiState = UiState.Success(data.recentSingingStreams ?: emptyLi
                    // --- START OF FIX ---
                    CarouselShelf<SingingStreamShelfItem>(
                        title = stringResource(R.string.shelf_title_recent_streams_fav
                        uiState = uiState,
                        actionContent = {
                            TextButton(onClick = {
                                videoListViewModel.setBrowseContextAndNavigate(org = "
                                navController.navigate(AppDestinations.HOME_ROUTE)
                            }) {
                                Text(stringResource(R.string.action_show_more))
                            }
                        },
                        itemContent = { item ->
                            val shell = item.video.toUnifiedDisplayItem(false, emptySe
                            UnifiedGridItem(
                                item = shell,
                                onClick = { navController.navigate(AppDestinations.vid
                            )
                        }
                    )
                    // --- END OF FIX ---
                }

                item {
                    val radios = remember {
                        data.recommended?.playlists?.filter {
                            it.type.startsWith("radio")
                        } ?: emptyList()
                    }
                    val uiState = UiState.Success(radios)
                    // --- START OF FIX ---
                    CarouselShelf<PlaylistStub>(
                        title = "Favorite Artist Radios",
                        uiState = uiState,
                        actionContent = {
                            TextButton(onClick = { /* TODO: Navigate to full list of f
                                Text(stringResource(R.string.action_show_more))
```

```
                                    }
                                },
                                itemContent = { item ->
                                    PlaylistCard(
                                        playlist = item,
                                        onPlaylistClicked = { navController.navigate(AppDestin
                                    )
                                }
                            )
                            // --- END OF FIX ---
                        }

                        item {
                            val recommendedChannels = remember { data.channels ?: emptyList()
                            val uiState = UiState.Success(recommendedChannels)
                            // --- START OF FIX ---
                            CarouselShelf<DiscoveryChannel>(
                                title = "Discover More Channels",
                                uiState = uiState,
                                actionContent = {
                                    TextButton(onClick = { /* TODO: Navigate to full list of f
                                        Text(stringResource(R.string.action_show_more))
                                    }
                                },
                                itemContent = { channel ->
                                    ChannelCard(
                                        channel = channel,
                                        onChannelClicked = { channelId -> navController.naviga
                                    )
                                }
                            )
                            // --- END OF FIX ---
                        }
                    }
                }
            }
        }
    }
}

// File: java\com\example\holodex\ui\screens\FullListViewScreen.kt
// File: java/com/example/holodex/ui/screens/FullListViewScreen.kt

package com.example.holodex.ui.screens

import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.PaddingValues
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.lazy.grid.GridCells
import androidx.compose.foundation.lazy.grid.GridItemSpan
import androidx.compose.foundation.lazy.grid.LazyVerticalGrid
```

```
import androidx.compose.foundation.lazy.grid.rememberLazyGridState
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.automirrored.filled.ArrowBack
import androidx.compose.material3.CircularProgressIndicator
import androidx.compose.material3.ExperimentalMaterial3Api
import androidx.compose.material3.Icon
import androidx.compose.material3.IconButton
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Text
import androidx.compose.material3.TopAppBar
import androidx.compose.runtime.Composable
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.runtime.derivedStateOf
import androidx.compose.runtime.getValue
import androidx.compose.runtime.remember
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.unit.dp
import androidx.hilt.navigation.compose.hiltViewModel
import androidx.lifecycle.compose.collectAsStateWithLifecycle
import androidx.media3.common.util.UnstableApi
import androidx.navigation.NavController
import com.example.holodex.R
import com.example.holodex.data.model.discovery.DiscoveryChannel
import com.example.holodex.data.model.discovery.PlaylistStub
import com.example.holodex.ui.AppDestinations
import com.example.holodex.ui.composables.ChannelCard
import com.example.holodex.ui.composables.PlaylistCard
import com.example.holodex.ui.composables.UnifiedGridItem
import com.example.holodex.viewmodel.DiscoveryViewModel
import com.example.holodex.viewmodel.FullListViewModel
import com.example.holodex.viewmodel.SubOrgHeader
import com.example.holodex.viewmodel.UnifiedDisplayItem
import com.example.holodex.viewmodel.VideoListViewModel.MusicCategoryType

@androidx.annotation.OptIn(UnstableApi::class)
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun FullListViewScreen(
    navController: NavController,
    categoryType: MusicCategoryType
) {
    val fullListViewModel: FullListViewModel = hiltViewModel()
    val discoveryViewModel: DiscoveryViewModel = hiltViewModel()

    val listState by fullListViewModel.listState.items.collectAsStateWithLifecycle()
    // --- START OF IMPLEMENTATION (1/3) ---
    val isLoadingMore by fullListViewModel.listState.isLoadingMore.collectAsStateWithLifecycle
    val endOfList by fullListViewModel.listState.endOfList.collectAsStateWithLifecycle()
    val gridState = rememberLazyGridState()

    // This derived state will be true when the user scrolls near the end of the list.
```

```
// It's a performant way to create a signal for loading more data.
val shouldLoadMore by remember {
    derivedStateOf {
        val layoutInfo = gridState.layoutInfo
        val totalItems = layoutInfo.totalItemsCount
        if (totalItems == 0) return@derivedStateOf false

        val lastVisibleItem = layoutInfo.visibleItemsInfo.lastOrNull()
            ?: return@derivedStateOf false

        // Trigger when the last visible item is within 10 items of the end
        lastVisibleItem.index >= totalItems - 10
    }
}

// This effect is triggered whenever the `shouldLoadMore` signal becomes true.
LaunchedEffect(shouldLoadMore) {
    if (shouldLoadMore && !isLoadingMore && !endOfList) {
        fullListViewModel.loadMore()
    }
}
// --- END OF IMPLEMENTATION (1/3) ---

Scaffold(
    topBar = {
        TopAppBar(
            title = { Text(categoryType.name.replace('_', ' ').lowercase().replaceFirstCha
            navigationIcon = {
                IconButton(onClick = { navController.popBackStack() }) {
                    Icon(Icons.AutoMirrored.Filled.ArrowBack, contentDescription = stringR
                }
            }
        )
    }
) { paddingValues ->
    LazyVerticalGrid(
        // --- START OF IMPLEMENTATION (2/3) ---
        state = gridState, // Assign the state to the grid
        // --- END OF IMPLEMENTATION (2/3) ---
        columns = GridCells.Adaptive(140.dp),
        modifier = Modifier.padding(paddingValues).fillMaxSize(),
        contentPadding = PaddingValues(16.dp),
        horizontalArrangement = Arrangement.spacedBy(12.dp),
        verticalArrangement = Arrangement.spacedBy(12.dp)
    ) {
        items(
            count = listState.size,
            key = { index ->
                val item = listState[index]
                when (item) {
                    is UnifiedDisplayItem -> item.stableId
                    is PlaylistStub -> item.id
                    is DiscoveryChannel -> item.id
                    is SubOrgHeader -> item.name
                    else -> item.hashCode()
```

```
                    }
                },
                span = { index ->
                    val item = listState[index]
                    if (item is SubOrgHeader) {
                        GridItemSpan(maxLineSpan)
                    } else {
                        GridItemSpan(1)
                    }
                }
            ) { index ->
                val item = listState[index]
                when (item) {
                    is UnifiedDisplayItem -> UnifiedGridItem(item = item, onClick = { discover
                    is PlaylistStub -> PlaylistCard(playlist = item, onPlaylistClicked = { pla
                        navController.navigate(AppDestinations.playlistDetailsRoute(playlistSt
                    is DiscoveryChannel -> ChannelCard(channel = item, onChannelClicked = { ch
                        navController.navigate("channel_details/$channelId")
                    })
                    is SubOrgHeader -> Text(
                        text = item.name,
                        style = MaterialTheme.typography.titleMedium,
                        fontWeight = FontWeight.Bold,
                        modifier = Modifier
                            .fillMaxWidth()
                            .padding(bottom = 8.dp, top = if (index == 0) 0.dp else 16.dp)
                    )
                }
            }

            // --- START OF IMPLEMENTATION (3/3) ---
            // Add a footer item to show the loading indicator when fetching the next page.
            if (isLoadingMore) {
                item(span = { GridItemSpan(maxLineSpan) }) {
                    Box(
                        modifier = Modifier
                            .fillMaxWidth()
                            .padding(vertical = 16.dp),
                        contentAlignment = Alignment.Center
                    ) {
                        CircularProgressIndicator(modifier = Modifier.size(36.dp))
                    }
                }
            }
            // --- END OF IMPLEMENTATION (3/3) ---
        }
    }
}

// File: java\com\example\holodex\ui\screens\HistoryScreen.kt
package com.example.holodex.ui.screens

import android.widget.Toast
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Column
```

```
import androidx.compose.foundation.layout.PaddingValues
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.automirrored.filled.PlaylistAdd
import androidx.compose.material.icons.filled.PlayArrow
import androidx.compose.material3.Button
import androidx.compose.material3.ButtonDefaults
import androidx.compose.material3.HorizontalDivider
import androidx.compose.material3.Icon
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.OutlinedButton
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.runtime.getValue
import androidx.compose.ui.Modifier
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.res.pluralStringResource
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.unit.dp
import androidx.hilt.navigation.compose.hiltViewModel
import androidx.lifecycle.compose.collectAsStateWithLifecycle
import androidx.media3.common.util.UnstableApi
import androidx.navigation.NavController
import com.example.holodex.R
import com.example.holodex.ui.composables.EmptyState
import com.example.holodex.ui.composables.UnifiedListItem
import com.example.holodex.viewmodel.FavoritesViewModel
import com.example.holodex.viewmodel.HistoryViewModel
import com.example.holodex.viewmodel.PlaylistManagementViewModel
import com.example.holodex.viewmodel.VideoListViewModel

@androidx.annotation.OptIn(UnstableApi::class)
@Composable
fun HistoryScreen(
    modifier: Modifier = Modifier,
    navController: NavController,
    videoListViewModel: VideoListViewModel,
    favoritesViewModel: FavoritesViewModel,
    playlistManagementViewModel: PlaylistManagementViewModel
) {
    val historyViewModel: HistoryViewModel = hiltViewModel()
    val historyItems by historyViewModel.unifiedHistoryItems.collectAsStateWithLifecycle()
    val context = LocalContext.current

    LaunchedEffect(Unit) {
        historyViewModel.transientMessage.collect { message ->
```

```
            Toast.makeText(context, message, Toast.LENGTH_SHORT).show()
        }
    }

    Column(modifier = modifier.fillMaxSize()) {
        if (historyItems.isEmpty()) {
            EmptyState(
                message = stringResource(R.string.message_no_history),
                onRefresh = {}
            )
        } else {
            HistoryHeader(
                songCount = historyItems.size,
                onPlayAll = { historyViewModel.playAllHistory() },
                onAddAllToQueue = { historyViewModel.addAllHistoryToQueue() }
            )
            HorizontalDivider()

            LazyColumn(
                modifier = Modifier.fillMaxSize(),
                contentPadding = PaddingValues(bottom = 80.dp)
            ) {
                items(
                    items = historyItems,
                    key = { item -> item.stableId } // FIX: Corrected typo from 'stableId'
                ) { item ->
                    UnifiedListItem(
                        item = item,
                        onItemClicked = { historyViewModel.playFromHistoryItem(item) },
                        videoListViewModel = videoListViewModel,
                        favoritesViewModel = favoritesViewModel,
                        playlistManagementViewModel = playlistManagementViewModel,
                        navController = navController
                    )
                }
            }
        }
    }
}

@Composable
private fun HistoryHeader(
    songCount: Int,
    onPlayAll: () -> Unit,
    onAddAllToQueue: () -> Unit
) {
    Column(
        modifier = Modifier
            .fillMaxWidth()
            .padding(16.dp),
        verticalArrangement = Arrangement.spacedBy(16.dp)
    ) {
        Column {
            Text(
                text = stringResource(id = R.string.recently_played_songs),
```

```
                style = MaterialTheme.typography.titleLarge,
                fontWeight = FontWeight.Bold
            )
            Text(
                text = pluralStringResource(
                    id = R.plurals.song_count_label,
                    count = songCount, // FIX: Pass count parameter
                    songCount
                ),
                style = MaterialTheme.typography.bodyMedium,
                color = MaterialTheme.colorScheme.onSurfaceVariant
            )
        }

        Row(
            modifier = Modifier.fillMaxWidth(),
            horizontalArrangement = Arrangement.spacedBy(8.dp)
        ) {
            Button(
                onClick = onPlayAll,
                modifier = Modifier.weight(1f)
            ) {
                Icon(Icons.Default.PlayArrow, contentDescription = null)
                Spacer(Modifier.size(ButtonDefaults.IconSpacing))
                Text(stringResource(id = R.string.action_play))
            }
            OutlinedButton(
                onClick = onAddAllToQueue,
                modifier = Modifier.weight(1f)
            ) {
                Icon(Icons.AutoMirrored.Filled.PlaylistAdd, contentDescription = null)
                Spacer(Modifier.size(ButtonDefaults.IconSpacing))
                Text(stringResource(id = R.string.action_add_to_queue))
            }
        }
    }
}

// File: java\com\example\holodex\ui\screens\HomeScreen.kt
package com.example.holodex.ui.screens

import android.widget.Toast
import androidx.activity.compose.BackHandler
import androidx.compose.animation.AnimatedVisibility
import androidx.compose.animation.fadeIn
import androidx.compose.animation.fadeOut
import androidx.compose.foundation.clickable
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.foundation.lazy.rememberLazyListState
```

```
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.automirrored.filled.ArrowBack
import androidx.compose.material.icons.filled.Clear
import androidx.compose.material.icons.filled.History
import androidx.compose.material.icons.filled.KeyboardArrowUp
import androidx.compose.material.icons.filled.Search
import androidx.compose.material.icons.filled.Settings
import androidx.compose.material.icons.filled.Tune
import androidx.compose.material3.Badge
import androidx.compose.material3.BadgedBox
import androidx.compose.material3.DockedSearchBar
import androidx.compose.material3.ExperimentalMaterial3Api
import androidx.compose.material3.FloatingActionButton
import androidx.compose.material3.Icon
import androidx.compose.material3.IconButton
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.ModalBottomSheet
import androidx.compose.material3.Scaffold
import androidx.compose.material3.SnackbarHost
import androidx.compose.material3.SnackbarHostState
import androidx.compose.material3.Text
import androidx.compose.material3.rememberModalBottomSheetState
import androidx.compose.runtime.Composable
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.runtime.derivedStateOf
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.rememberCoroutineScope
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.text.style.TextOverflow
import androidx.compose.ui.unit.dp
import androidx.hilt.navigation.compose.hiltViewModel
import androidx.lifecycle.compose.collectAsStateWithLifecycle
import androidx.media3.common.util.UnstableApi
import androidx.navigation.NavController
import com.example.holodex.R
import com.example.holodex.ui.AppDestinations
import com.example.holodex.ui.composables.CustomPagedUnifiedList
import com.example.holodex.ui.composables.EmptyState
import com.example.holodex.ui.composables.LoadingSkeleton
import com.example.holodex.ui.composables.sheets.BrowseFiltersSheet
import com.example.holodex.viewmodel.FavoritesViewModel
import com.example.holodex.viewmodel.PlaylistManagementViewModel
import com.example.holodex.viewmodel.VideoListViewModel
import com.example.holodex.viewmodel.VideoListViewModel.MusicCategoryType
import kotlinx.coroutines.flow.collectLatest
import kotlinx.coroutines.launch
import timber.log.Timber

@UnstableApi
```

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun HomeScreen(
    navController: NavController,
    videoListViewModel: VideoListViewModel,
    playlistManagementViewModel: PlaylistManagementViewModel,
) {
    val favoritesViewModel: FavoritesViewModel = hiltViewModel()

    val coroutineScope = rememberCoroutineScope()
    val snackbarHostState = remember { SnackbarHostState() }
    val listState = rememberLazyListState()

    val activeContext by videoListViewModel.activeListContextType.collectAsStateWithLifecycle(
    val searchQuery by videoListViewModel.currentSearchQuery.collectAsStateWithLifecycle()
    val isSearchActive by videoListViewModel.isSearchActive.collectAsStateWithLifecycle()
    val searchHistory by videoListViewModel.searchHistory.collectAsStateWithLifecycle()
    val browseFilters by videoListViewModel.browseFilterState.collectAsStateWithLifecycle()

    val browseItems by videoListViewModel.browseListState.collectAsStateWithLifecycle()
    val browseIsLoading by videoListViewModel.browseIsLoadingInitial.collectAsStateWithLifecyc
    val browseIsLoadingMore by videoListViewModel.browseIsLoadingMore.collectAsStateWithLifecy
    val browseEndOfList by videoListViewModel.browseEndOfList.collectAsStateWithLifecycle()
    val browseIsRefreshing by videoListViewModel.browseIsRefreshing.collectAsStateWithLifecycl

    val searchItems by videoListViewModel.searchListState.collectAsStateWithLifecycle()
    val searchIsLoading by videoListViewModel.searchIsLoadingInitial.collectAsStateWithLifecyc
    val searchIsLoadingMore by videoListViewModel.searchIsLoadingMore.collectAsStateWithLifecy
    val searchEndOfList by videoListViewModel.searchEndOfList.collectAsStateWithLifecycle()

    var showFilterSheet by remember { mutableStateOf(false) }

    LaunchedEffect(Unit) {
        videoListViewModel.initializeAndFetch()
    }
    val context = LocalContext.current
    LaunchedEffect(Unit) {
        videoListViewModel.transientMessage.collectLatest { message ->
            Toast.makeText(context, message, Toast.LENGTH_SHORT).show()
        }
    }
    BackHandler(enabled = isSearchActive) {
        videoListViewModel.clearSearchAndReturnToBrowse()
    }

    Box(Modifier.fillMaxSize()) {
        Scaffold(
            snackbarHost = { SnackbarHost(snackbarHostState) },
            floatingActionButton = {
                val showFab by remember { derivedStateOf { listState.firstVisibleItemIndex > 5
                AnimatedVisibility(
                    visible = showFab && !isSearchActive,
                    enter = fadeIn(),
                    exit = fadeOut()
                ) {
```

```
                FloatingActionButton(onClick = {
                    coroutineScope.launch {
                        listState.animateScrollToItem(
                            0
                        )
                    }
                }) {
                    Icon(Icons.Filled.KeyboardArrowUp, stringResource(R.string.scroll_to_t
                }
            }
        }
    ) { innerPadding ->
        Box(modifier = Modifier
            .padding(innerPadding)
            .padding(top = 80.dp)) {
            if (activeContext == MusicCategoryType.SEARCH) {
                // --- SEARCH CONTENT ---
                if (searchIsLoading && searchItems.isEmpty()) {
                    LoadingSkeleton(modifier = Modifier.fillMaxSize())
                } else if (searchItems.isEmpty() && searchEndOfList && !searchIsLoading) {
                    EmptyState(
                        message = stringResource(
                            R.string.status_search_no_results,
                            searchQuery
                        ), onRefresh = { videoListViewModel.refreshCurrentListViaPull() })
                } else {
                    CustomPagedUnifiedList(
                        listKeyPrefix = "home_search",
                        items = searchItems,
                        listState = listState,
                        onItemClicked = { item ->
                            // --- LOGGING POINT 3A: SEARCH LIST CLICK ---
                            Timber.d("HomeScreen (Search): Click received for item '${item
                            videoListViewModel.onVideoClicked(item)
                        }, videoListViewModel = videoListViewModel,
                        playlistManagementViewModel = playlistManagementViewModel,
                        navController = navController,
                        favoritesViewModel = favoritesViewModel,
                        isLoadingMore = searchIsLoadingMore,
                        endOfList = searchEndOfList,
                        onLoadMore = { videoListViewModel.loadMore(MusicCategoryType.SEARC
                        isRefreshing = false, // Search list doesn't use pull-to-refresh
                        onRefresh = {}

                    )
                }
            } else {
                // --- BROWSE CONTENT ---
                if (browseIsLoading && browseItems.isEmpty()) {
                    LoadingSkeleton(modifier = Modifier.fillMaxSize())
                } else {
                    CustomPagedUnifiedList(
                        listKeyPrefix = "home_browse",
                        items = browseItems,
                        listState = listState,
```

```
                    onItemClicked = { item ->
                        videoListViewModel.onVideoClicked(item)
                    }, videoListViewModel = videoListViewModel,
                    favoritesViewModel = favoritesViewModel,
                    playlistManagementViewModel = playlistManagementViewModel,
                    isLoadingMore = browseIsLoadingMore,
                    endOfList = browseEndOfList,
                    navController = navController,
                    onLoadMore = { videoListViewModel.loadMore(activeContext) },
                    isRefreshing = browseIsRefreshing,
                    onRefresh = { videoListViewModel.refreshCurrentListViaPull() }
                )
            }
        }
    }
}

DockedSearchBar(
    modifier = Modifier
        .align(Alignment.TopCenter)
        .padding(top = 8.dp, start = 16.dp, end = 16.dp)
        .fillMaxWidth(),
    query = searchQuery,
    onQueryChange = { query -> videoListViewModel.onSearchQueryChange(query) },
    onSearch = { query -> videoListViewModel.performSearch(query) },
    active = isSearchActive,
    onActiveChange = { active -> videoListViewModel.setSearchActive(active) },
    placeholder = {
        Text(
            stringResource(R.string.search_your_music_hint),
            maxLines = 1,
            overflow = TextOverflow.Ellipsis
        )
    },
    leadingIcon = {
        if (isSearchActive) {
            IconButton(onClick = { videoListViewModel.clearSearchAndReturnToBrowse() }
                Icon(Icons.AutoMirrored.Filled.ArrowBack, "Back")
            }
        } else {
            Icon(Icons.Filled.Search, "Search Icon")
        }
    },
    trailingIcon = {
        Row {
            if (isSearchActive && searchQuery.isNotEmpty()) {
                IconButton(onClick = { videoListViewModel.onSearchQueryChange("") }) {
                    Icon(Icons.Filled.Clear, stringResource(R.string.action_clear_sear
                }
            }
            if (!isSearchActive) {
                IconButton(onClick = { showFilterSheet = true }) {
                    BadgedBox(badge = {
                        if (browseFilters.hasActiveFilters) {
                            Badge { Text(browseFilters.activeFilterCount.toString()) }
```

```
                            }
                        }) {
                            Icon(Icons.Filled.Tune, stringResource(R.string.action_filter_
                        }
                    }
                    IconButton(onClick = { navController.navigate(AppDestinations.SETTINGS
                        Icon(Icons.Filled.Settings, stringResource(R.string.settings_title
                    }
                }
            }
        ) {
            SearchHistoryList(
                searchHistory = searchHistory,
                onHistoryItemClick = { query ->
                    videoListViewModel.onSearchQueryChange(query)
                    videoListViewModel.performSearch(query)
                }
            )
        }
    }

    if (showFilterSheet) {
        ModalBottomSheet(
            onDismissRequest = { showFilterSheet = false },
            sheetState = rememberModalBottomSheetState(skipPartiallyExpanded = true)
        ) {
            BrowseFiltersSheet(
                initialFilters = browseFilters,
                onFiltersApplied = { newFilters ->
                    showFilterSheet = false
                    videoListViewModel.updateBrowseFilters(newFilters)
                },
                onDismiss = { showFilterSheet = false }
            )
        }
    }
}


@Composable
private fun SearchHistoryList(
    searchHistory: List<String>,
    onHistoryItemClick: (String) -> Unit
) {
    if (searchHistory.isEmpty()) {
        Box(
            modifier = Modifier
                .fillMaxSize()
                .padding(16.dp), contentAlignment = Alignment.Center
        ) {
            Text(
                "No recent searches",
                style = MaterialTheme.typography.bodyLarge,
                color = MaterialTheme.colorScheme.onSurfaceVariant
```

```
            )
        }
        return
    }

    LazyColumn(modifier = Modifier.fillMaxSize()) {
        item {
            Text(
                text = stringResource(R.string.search_history_title),
                style = MaterialTheme.typography.titleSmall,
                modifier = Modifier.padding(horizontal = 16.dp, vertical = 8.dp)
            )
        }
        items(searchHistory) { query ->
            Row(
                modifier = Modifier
                    .fillMaxWidth()
                    .clickable { onHistoryItemClick(query) }
                    .padding(horizontal = 16.dp, vertical = 12.dp),
                verticalAlignment = Alignment.CenterVertically
            ) {
                Icon(Icons.Filled.History, null, modifier = Modifier.padding(end = 16.dp))
                Text(query, style = MaterialTheme.typography.bodyLarge)
            }
        }
    }
}
```

```
// File: java\com\example\holodex\ui\screens\LibraryScreen.kt
// File: java/com/example/holodex/ui/screens/LibraryScreen.kt
package com.example.holodex.ui.screens

import androidx.compose.animation.animateColorAsState
import androidx.compose.animation.core.FastOutSlowInEasing
import androidx.compose.animation.core.animateDpAsState
import androidx.compose.animation.core.tween
import androidx.compose.foundation.ExperimentalFoundationApi
import androidx.compose.foundation.background
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.BoxWithConstraints
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.offset
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.width
import androidx.compose.foundation.pager.HorizontalPager
import androidx.compose.foundation.pager.PagerState
import androidx.compose.foundation.pager.rememberPagerState
import androidx.compose.foundation.shape.RoundedCornerShape
```

```
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.automirrored.filled.ViewList
import androidx.compose.material.icons.filled.Add
import androidx.compose.material.icons.filled.GridView
import androidx.compose.material3.ExperimentalMaterial3Api
import androidx.compose.material3.FloatingActionButton
import androidx.compose.material3.Icon
import androidx.compose.material3.IconButton
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Surface
import androidx.compose.material3.Text
import androidx.compose.material3.TopAppBar
import androidx.compose.material3.TopAppBarDefaults
import androidx.compose.runtime.Composable
import androidx.compose.runtime.derivedStateOf
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.rememberCoroutineScope
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.unit.dp
import androidx.hilt.navigation.compose.hiltViewModel
import androidx.lifecycle.compose.collectAsStateWithLifecycle
import androidx.navigation.NavController
import com.example.holodex.R
import com.example.holodex.ui.AppDestinations
import com.example.holodex.ui.dialogs.AddExternalChannelDialog
import com.example.holodex.viewmodel.ExternalChannelViewModel
import com.example.holodex.viewmodel.PlaylistManagementViewModel
import kotlinx.coroutines.launch

private enum class LibraryTab(val titleRes: Int) {
    PLAYLISTS(R.string.bottom_nav_playlists),
    FAVORITES(R.string.bottom_nav_favorites),
    HISTORY(R.string.screen_title_history)
}

@OptIn(ExperimentalMaterial3Api::class, ExperimentalFoundationApi::class)
@Composable
fun LibraryScreen(
    navController: NavController,
    playlistManagementViewModel: PlaylistManagementViewModel,
) {
    val pagerState = rememberPagerState(initialPage = 1, pageCount = { LibraryTab.entries.size
    val coroutineScope = rememberCoroutineScope()
    var isGridView by remember { mutableStateOf(false) }

    val externalChannelViewModel: ExternalChannelViewModel = hiltViewModel()
```

```
val showAddChannelDialog by externalChannelViewModel.showDialog.collectAsStateWithLifecycl

if (showAddChannelDialog) {
    AddExternalChannelDialog(onDismissRequest = { externalChannelViewModel.closeDialog() }
}

Scaffold(
    topBar = {
        LibraryTopAppBar(
            isGridView = isGridView,
            onViewToggle = { isGridView = !isGridView }
        )
    },
    floatingActionButton = {
        FloatingActionButton(onClick = { externalChannelViewModel.openDialog() }) {
            Icon(Icons.Default.Add, contentDescription = "Add External Channel")
        }
    }
) { paddingValues ->
    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(paddingValues)
    ) {
        AnimatedCustomTabRow(
            selectedTabIndex = pagerState.currentPage,
            tabs = LibraryTab.entries.map { stringResource(it.titleRes) },
            pagerState = pagerState,
            onTabSelected = { index ->
                coroutineScope.launch { pagerState.animateScrollToPage(index) }
            }
        )

        HorizontalPager(state = pagerState, modifier = Modifier.fillMaxSize()) { page ->
            when (LibraryTab.entries[page]) {
                LibraryTab.FAVORITES -> FavoritesTab(
                    isGridView = isGridView,
                    navController = navController,
                    playlistManagementViewModel = playlistManagementViewModel
                )
                LibraryTab.PLAYLISTS -> PlaylistsTab(
                    onPlaylistClicked = { playlist ->
                        val idToNavigate = when {
                            playlist.playlistId < 0 && playlist.serverId == null -> playli
                            playlist.playlistId > 0 -> playlist.playlistId.toString()
                            else -> playlist.serverId
                        }
                        if (!idToNavigate.isNullOrBlank()) {
                            navController.navigate(AppDestinations.playlistDetailsRoute(id
                        }
                    }
                )
                LibraryTab.HISTORY -> HistoryTab(
                    navController = navController,
                    playlistManagementViewModel = playlistManagementViewModel
```

```
                    )
                }
            }
        }
    }
}


@OptIn(ExperimentalMaterial3Api::class)
@Composable
private fun LibraryTopAppBar(
    isGridView: Boolean,
    onViewToggle: () -> Unit
) {
    TopAppBar(
        title = {
            Text(
                text = stringResource(R.string.bottom_nav_library),
                style = MaterialTheme.typography.headlineSmall,
                fontWeight = FontWeight.Bold
            )
        },
        actions = {
            IconButton(onClick = onViewToggle) {
                Icon(
                    imageVector = if (isGridView) Icons.AutoMirrored.Filled.ViewList else Icon
                    contentDescription = stringResource(if (isGridView) R.string.action_view_a
                )
            }
        },
        colors = TopAppBarDefaults.topAppBarColors(
            containerColor = MaterialTheme.colorScheme.surface,
            titleContentColor = MaterialTheme.colorScheme.onSurface
        )
    )
}

@OptIn(ExperimentalFoundationApi::class)
@Composable
private fun AnimatedCustomTabRow(
    selectedTabIndex: Int,
    tabs: List<String>,
    pagerState: PagerState,
    onTabSelected: (Int) -> Unit,
    modifier: Modifier = Modifier,
    containerColor: Color = MaterialTheme.colorScheme.surface,
    indicatorColor: Color = MaterialTheme.colorScheme.primary
) {
    BoxWithConstraints(
        modifier = modifier
            .fillMaxWidth()
            .background(containerColor)
            .padding(horizontal = 16.dp, vertical = 8.dp)
    ) {
        val tabWidth = this@BoxWithConstraints.maxWidth / tabs.size
```

```
        Box(
            modifier = Modifier
                .fillMaxWidth()
                .height(48.dp)
                .background(
                    color = MaterialTheme.colorScheme.surfaceVariant.copy(alpha = 0.3f),
                    shape = RoundedCornerShape(24.dp)
                )
        )
        val targetOffset = tabWidth * selectedTabIndex
        val pagerOffset = tabWidth * pagerState.currentPageOffsetFraction
        val indicatorOffset = targetOffset + pagerOffset
        val animatedIndicatorOffset by animateDpAsState(
            targetValue = indicatorOffset,
            animationSpec = tween(durationMillis = 250, easing = FastOutSlowInEasing),
            label = "indicator_offset"
        )
        Box(
            modifier = Modifier
                .offset(x = animatedIndicatorOffset)
                .width(tabWidth)
                .height(48.dp)
                .padding(4.dp)
                .background(color = indicatorColor, shape = RoundedCornerShape(20.dp))
        )
        Row(modifier = Modifier.fillMaxWidth(), horizontalArrangement = Arrangement.SpaceEvenl
            tabs.forEachIndexed { index, title ->
                AnimatedTab(
                    title = title,
                    selected = selectedTabIndex == index,
                    onClick = { onTabSelected(index) },
                    modifier = Modifier.weight(1f)
                )
            }
        }
    }
}

@Composable
private fun AnimatedTab(
    title: String,
    selected: Boolean,
    onClick: () -> Unit,
    modifier: Modifier = Modifier
) {
    val animatedColor by animateColorAsState(
        targetValue = if (selected) MaterialTheme.colorScheme.onPrimary else MaterialTheme.col
        animationSpec = tween(durationMillis = 150, easing = FastOutSlowInEasing),
        label = "tab_color"
    )
    val animatedFontWeight by remember { derivedStateOf { if (selected) FontWeight.Bold else F
    Surface(
        modifier = modifier
            .height(48.dp)
            .clip(RoundedCornerShape(24.dp)),
```

```
        onClick = onClick,
        color = Color.Transparent,
        contentColor = animatedColor,
        shape = RoundedCornerShape(24.dp)
    ) {
        Box(contentAlignment = Alignment.Center, modifier = Modifier.fillMaxSize()) {
            Text(text = title, color = animatedColor, style = MaterialTheme.typography.titleSm
        }
    }
}

@Composable
private fun PlaylistsTab(onPlaylistClicked: (com.example.holodex.data.db.PlaylistEntity) -> Un
    PlaylistsScreen(
        modifier = Modifier.fillMaxSize(),
        playlistManagementViewModel = hiltViewModel(),
        onPlaylistClicked = onPlaylistClicked
    )
}

@Composable
private fun FavoritesTab(
    isGridView: Boolean,
    navController: NavController,
    playlistManagementViewModel: PlaylistManagementViewModel
) {
    FavoritesScreen(
        isGridView = isGridView,
        modifier = Modifier.fillMaxSize(),
        videoListViewModel = hiltViewModel(),
        favoritesViewModel = hiltViewModel(),
        playlistManagementViewModel = playlistManagementViewModel,
        navController = navController
    )
}

@Composable
private fun HistoryTab(
    navController: NavController,
    playlistManagementViewModel: PlaylistManagementViewModel
) {
    HistoryScreen(
        modifier = Modifier.fillMaxSize(),
        navController = navController,
        videoListViewModel = hiltViewModel(),
        favoritesViewModel = hiltViewModel(),
        playlistManagementViewModel = playlistManagementViewModel
    )
}

// File: java\com\example\holodex\ui\screens\PlaylistDetailsScreen.kt
@file:kotlin.OptIn(ExperimentalMaterial3Api::class, ExperimentalFoundationApi::class)

package com.example.holodex.ui.screens
```

```
import android.widget.Toast
import androidx.annotation.OptIn
import androidx.compose.foundation.BorderStroke
import androidx.compose.foundation.ExperimentalFoundationApi
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.PaddingValues
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.itemsIndexed
import androidx.compose.foundation.lazy.rememberLazyListState
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.automirrored.filled.ArrowBack
import androidx.compose.material.icons.automirrored.filled.PlaylistAdd
import androidx.compose.material.icons.automirrored.filled.PlaylistPlay
import androidx.compose.material.icons.filled.Cancel
import androidx.compose.material.icons.filled.Check
import androidx.compose.material.icons.filled.Edit
import androidx.compose.material3.Button
import androidx.compose.material3.ButtonDefaults
import androidx.compose.material3.ExperimentalMaterial3Api
import androidx.compose.material3.HorizontalDivider
import androidx.compose.material3.Icon
import androidx.compose.material3.IconButton
import androidx.compose.material3.LocalContentColor
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.OutlinedButton
import androidx.compose.material3.Scaffold
import androidx.compose.material3.SnackbarDuration
import androidx.compose.material3.SnackbarHost
import androidx.compose.material3.SnackbarHostState
import androidx.compose.material3.Text
import androidx.compose.material3.TopAppBar
import androidx.compose.material3.TopAppBarDefaults
import androidx.compose.runtime.Composable
import androidx.compose.runtime.CompositionLocalProvider
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.runtime.derivedStateOf
import androidx.compose.runtime.getValue
import androidx.compose.runtime.remember
import androidx.compose.runtime.rememberCoroutineScope
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.hapticfeedback.HapticFeedbackType
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.platform.LocalHapticFeedback
import androidx.compose.ui.res.painterResource
```

```
import androidx.compose.ui.res.pluralStringResource
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.text.style.TextAlign
import androidx.compose.ui.text.style.TextOverflow
import androidx.compose.ui.unit.dp
import androidx.hilt.navigation.compose.hiltViewModel
import androidx.lifecycle.compose.collectAsStateWithLifecycle
import androidx.media3.common.util.UnstableApi
import androidx.navigation.NavController
import com.example.holodex.R
import com.example.holodex.data.db.PlaylistEntity
import com.example.holodex.data.model.discovery.PlaylistStub
import com.example.holodex.ui.composables.EmptyState
import com.example.holodex.ui.composables.ErrorStateWithRetry
import com.example.holodex.ui.composables.LoadingState
import com.example.holodex.ui.composables.SimpleProcessedBackground
import com.example.holodex.ui.composables.UnifiedListItem
import com.example.holodex.util.ArtworkResolver
import com.example.holodex.util.DynamicTheme
import com.example.holodex.util.findActivity
import com.example.holodex.viewmodel.FavoritesViewModel
import com.example.holodex.viewmodel.PlaylistDetailsViewModel
import com.example.holodex.viewmodel.PlaylistManagementViewModel
import com.example.holodex.viewmodel.UnifiedDisplayItem
import com.example.holodex.viewmodel.VideoListViewModel
import kotlinx.coroutines.launch
import sh.calvin.reorderable.ReorderableItem
import sh.calvin.reorderable.rememberReorderableLazyListState

@OptIn(UnstableApi::class)
@Composable
fun PlaylistDetailsScreen(
    navController: NavController,
    onNavigateUp: () -> Unit,
    playlistManagementViewModel: PlaylistManagementViewModel
) {
    val playlistDetailsViewModel: PlaylistDetailsViewModel = hiltViewModel()

    val playlistDetails by playlistDetailsViewModel.playlistDetails.collectAsStateWithLifecycl
    val items by playlistDetailsViewModel.unifiedPlaylistItems.collectAsStateWithLifecycle()
    val isLoading by playlistDetailsViewModel.isLoading.collectAsStateWithLifecycle()
    val error by playlistDetailsViewModel.error.collectAsStateWithLifecycle()
    val isEditMode by playlistDetailsViewModel.isEditMode.collectAsStateWithLifecycle()
    val editablePlaylist by playlistDetailsViewModel.editablePlaylist.collectAsStateWithLifecy
    val isPlaylistOwned by playlistDetailsViewModel.isPlaylistOwned.collectAsStateWithLifecycl
    val dynamicTheme by playlistDetailsViewModel.dynamicTheme.collectAsStateWithLifecycle()

    val snackbarHostState = remember { SnackbarHostState() }
    val coroutineScope = rememberCoroutineScope()
    val context = LocalContext.current

    val videoListViewModel: VideoListViewModel = hiltViewModel(findActivity())
    val favoritesViewModel: FavoritesViewModel = hiltViewModel()
```

```
val backgroundImageUrl by remember(items, playlistDetails) {
    derivedStateOf {
        items.firstOrNull()?.artworkUrls?.firstOrNull()
            ?: playlistDetails?.let {
                val stub = PlaylistStub(it.serverId ?: it.playlistId.toString(), it.name ?
                ArtworkResolver.getPlaylistArtworkUrl(stub)
            }
    }
}

LaunchedEffect(error) {
    error?.let {
        coroutineScope.launch {
            snackbarHostState.showSnackbar(message = it, duration = SnackbarDuration.Long)
            playlistDetailsViewModel.clearError()
        }
    }
}

LaunchedEffect(Unit) {
    playlistDetailsViewModel.transientMessage.collect { message ->
        Toast.makeText(context, message, Toast.LENGTH_SHORT).show()
    }
}

Scaffold(
    snackbarHost = { SnackbarHost(snackbarHostState) },
    topBar = {
        TopAppBar(
            title = {
                if (!isEditMode) {
                    Text(
                        text = playlistDetails?.name ?: "",
                        maxLines = 1,
                        overflow = TextOverflow.Ellipsis
                    )
                }
            },
            navigationIcon = {
                IconButton(onClick = onNavigateUp) {
                    Icon(
                        Icons.AutoMirrored.Filled.ArrowBack,
                        contentDescription = stringResource(R.string.action_back)
                    )
                }
            },
            actions = {
                if (isEditMode) {
                    IconButton(onClick = { playlistDetailsViewModel.cancelEditMode() }) {
                        Icon(Icons.Default.Cancel, contentDescription = "Cancel Edit")
                    }
                    IconButton(onClick = { playlistDetailsViewModel.saveChanges() }) {
                        Icon(Icons.Default.Check, contentDescription = "Save Changes")
                    }
                } else {
```

```
                    if (isPlaylistOwned) {
                        IconButton(onClick = { playlistDetailsViewModel.enterEditMode() })
                            Icon(Icons.Default.Edit, contentDescription = "Edit Playlist")
                        }
                    }
                    val isShuffleActive by playlistDetailsViewModel.isPlaylistShuffleActiv
                    IconButton(onClick = { playlistDetailsViewModel.togglePlaylistShuffleM
                        Icon(
                            painter = painterResource(id = if (isShuffleActive) R.drawable
                            contentDescription = stringResource(R.string.action_shuffle),
                            tint = if (isShuffleActive) dynamicTheme.primary else dynamicT
                        )
                    }
                }
            },
            colors = TopAppBarDefaults.topAppBarColors(
                containerColor = Color.Transparent,
                titleContentColor = dynamicTheme.onPrimary,
                navigationIconContentColor = dynamicTheme.onPrimary,
                actionIconContentColor = dynamicTheme.onPrimary
            )
        )
    }
) { paddingValues ->
    Box(modifier = Modifier.fillMaxSize()) {
        SimpleProcessedBackground(
            artworkUri = backgroundImageUrl,
            dynamicColor = dynamicTheme.primary
        )
        CompositionLocalProvider(LocalContentColor provides dynamicTheme.onPrimary) {
            Box(modifier = Modifier.padding(paddingValues).fillMaxSize()) {
                when {
                    isLoading && items.isEmpty() -> {
                        LoadingState(message = stringResource(R.string.loading_content_mes
                    }
                    error != null && items.isEmpty() -> {
                        ErrorStateWithRetry(
                            message = error!!,
                            onRetry = { playlistDetailsViewModel.loadPlaylistDetails() }
                        )
                    }
                    items.isEmpty() && !isLoading -> {
                        EmptyState(
                            message = stringResource(R.string.message_playlist_is_empty),
                            onRefresh = { playlistDetailsViewModel.loadPlaylistDetails() }
                        )
                    }
                    else -> {
                        PlaylistContent(
                            items = items,
                            playlistDetails = if (isEditMode) editablePlaylist else playli
                            isEditMode = isEditMode,
                            navController = navController,
                            videoListViewModel = videoListViewModel,
                            favoritesViewModel = favoritesViewModel,
```

```
                                playlistManagementViewModel = playlistManagementViewModel,
                                playlistDetailsViewModel = playlistDetailsViewModel,
                                dynamicTheme = dynamicTheme
                            )
                        }
                    }
                }
            }
        }
    }
}

@UnstableApi
@Composable
private fun PlaylistContent(
    items: List<UnifiedDisplayItem>,
    playlistDetails: PlaylistEntity?,
    isEditMode: Boolean,
    navController: NavController,
    videoListViewModel: VideoListViewModel,
    favoritesViewModel: FavoritesViewModel,
    playlistManagementViewModel: PlaylistManagementViewModel,
    playlistDetailsViewModel: PlaylistDetailsViewModel,
    dynamicTheme: DynamicTheme
) {
    val lazyListState = rememberLazyListState()
    val reorderableState = rememberReorderableLazyListState(
        lazyListState = lazyListState,
        onMove = { from, to ->
            playlistDetailsViewModel.reorderItemInEditMode(from.index, to.index)
        }
    )

    LazyColumn(state = lazyListState, modifier = Modifier.fillMaxSize(), contentPadding = Padd
        item {
            if (isEditMode && playlistDetails != null) {
                EditablePlaylistHeader(
                    playlist = playlistDetails,
                    onNameChange = { playlistDetailsViewModel.updateDraftName(it) },
                    onDescriptionChange = { playlistDetailsViewModel.updateDraftDescription(it
                )
            } else {
                PlaylistHeader(
                    playlist = playlistDetails,
                    itemCount = items.size,
                    onPlayAll = { playlistDetailsViewModel.playAllItemsInPlaylist() },
                    onAddAllToQueue = { playlistDetailsViewModel.addAllToQueue() },
                    dynamicTheme = dynamicTheme
                )
            }
            HorizontalDivider(color = LocalContentColor.current.copy(alpha = 0.2f))
        }

        itemsIndexed(
            items = items,
```

```
            key = { _, item -> item.stableId }
        ) { index, item ->
            ReorderableItem(state = reorderableState, key = item.stableId, enabled = isEditMod
                UnifiedListItem(
                    item = item,
                    onItemClicked = { if (!isEditMode) playlistDetailsViewModel.playFromItem(i
                    navController = navController,
                    videoListViewModel = videoListViewModel,
                    favoritesViewModel = favoritesViewModel,
                    playlistManagementViewModel = playlistManagementViewModel,
                    isEditing = isEditMode,
                    onRemoveClicked = { playlistDetailsViewModel.removeItemInEditMode(item) },
                    dragHandleModifier = Modifier.longPressDraggableHandle()
                )
            }
        }
    }
}

@Composable
private fun PlaylistHeader(
    playlist: PlaylistEntity?,
    itemCount: Int,
    onPlayAll: () -> Unit,
    onAddAllToQueue: () -> Unit,
    dynamicTheme: DynamicTheme
) {
    val haptic = LocalHapticFeedback.current

    Column(
        modifier = Modifier
            .fillMaxWidth()
            .padding(16.dp),
        verticalArrangement = Arrangement.spacedBy(8.dp),
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        playlist?.let {
            Text(
                text = it.name ?: "Untitled Playlist",
                style = MaterialTheme.typography.headlineSmall,
                fontWeight = FontWeight.Bold,
                textAlign = TextAlign.Center
            )
            val description = it.description
            if (!description.isNullOrBlank()) {
                Text(
                    text = description,
                    style = MaterialTheme.typography.bodyMedium,
                    textAlign = TextAlign.Center
                )
            }
        }
        Text(
            text = pluralStringResource(R.plurals.item_count, itemCount, itemCount),
            style = MaterialTheme.typography.labelMedium
```

```
        )

        Spacer(modifier = Modifier.height(8.dp))

        if (itemCount > 0) {
            ActionButtons(
                onPlayAll = {
                    onPlayAll()
                    haptic.performHapticFeedback(HapticFeedbackType.LongPress)
                },
                onAddAllToQueue = {
                    onAddAllToQueue()
                    haptic.performHapticFeedback(HapticFeedbackType.LongPress)
                },
                dynamicTheme = dynamicTheme
            )
        }
    }
}

@Composable
private fun ActionButtons(
    onPlayAll: () -> Unit,
    onAddAllToQueue: () -> Unit,
    dynamicTheme: DynamicTheme
) {
    Row(modifier = Modifier.fillMaxWidth(), horizontalArrangement = Arrangement.spacedBy(8.dp)
        Button(
            onClick = onPlayAll,
            modifier = Modifier.weight(1f),
            colors = ButtonDefaults.buttonColors(
                containerColor = dynamicTheme.onPrimary,
                contentColor = dynamicTheme.primary
            )
        ) {
            Icon(Icons.AutoMirrored.Filled.PlaylistPlay, contentDescription = null, modifier =
            Spacer(Modifier.size(ButtonDefaults.IconSpacing))
            Text(stringResource(R.string.action_play_all))
        }

        OutlinedButton(
            onClick = onAddAllToQueue,
            modifier = Modifier.weight(1f),
            colors = ButtonDefaults.outlinedButtonColors(contentColor = dynamicTheme.onPrimary
            border = BorderStroke(1.dp, dynamicTheme.onPrimary.copy(alpha = 0.5f))
        ) {
            Icon(Icons.AutoMirrored.Filled.PlaylistAdd, contentDescription = null, modifier =
            Spacer(Modifier.size(ButtonDefaults.IconSpacing))
            Text(stringResource(id = R.string.action_add_to_queue))
        }
    }
}

// File: java\com\example\holodex\ui\screens\PlaylistsScreen.kt
// File: java/com/example/holodex/ui/screens/PlaylistsScreen.kt
```

```
// --- REFACTORED AND RENAMED ---
package com.example.holodex.ui.screens

import androidx.compose.foundation.clickable
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.PaddingValues
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.automirrored.filled.PlaylistPlay
import androidx.compose.material.icons.filled.Delete
import androidx.compose.material3.Button
import androidx.compose.material3.ExperimentalMaterial3Api
import androidx.compose.material3.HorizontalDivider
import androidx.compose.material3.Icon
import androidx.compose.material3.IconButton
import androidx.compose.material3.ListItem
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.text.style.TextOverflow
import androidx.compose.ui.unit.dp
import androidx.lifecycle.compose.collectAsStateWithLifecycle
import com.example.holodex.R
import com.example.holodex.data.db.PlaylistEntity
import com.example.holodex.ui.dialogs.CreatePlaylistDialog
import com.example.holodex.viewmodel.PlaylistManagementViewModel

@Composable
fun PlaylistsScreen(
    modifier: Modifier = Modifier,
    playlistManagementViewModel: PlaylistManagementViewModel,
    onPlaylistClicked: (PlaylistEntity) -> Unit
) {
    val playlists: List<PlaylistEntity> by playlistManagementViewModel.allDisplayablePlaylists
    val showCreateDialog by playlistManagementViewModel.showCreatePlaylistDialog.collectAsStat

    if (showCreateDialog) {
        CreatePlaylistDialog(
            onDismissRequest = { playlistManagementViewModel.closeCreatePlaylistDialog() },
            onCreatePlaylist = { name, description ->
                playlistManagementViewModel.confirmCreatePlaylist(name, description)
            }
        )
    }
```

```
    Box(modifier = modifier.fillMaxSize()) {
        if (playlists.isEmpty()) {
            Box(modifier = Modifier.fillMaxSize(), contentAlignment = Alignment.Center) {
                Column(horizontalAlignment = Alignment.CenterHorizontally) {
                    Text(stringResource(R.string.message_no_playlists_yet))
                    Spacer(Modifier.height(8.dp))
                    Button(onClick = { playlistManagementViewModel.openCreatePlaylistDialog()
                        Text(stringResource(R.string.action_create_playlist))
                    }
                }
            }
        } else {
            LazyColumn(
                contentPadding = PaddingValues(vertical = 8.dp)
            ) {
                items(playlists, key = { it.playlistId }) { playlist ->
                    PlaylistItemRow(
                        playlist = playlist,
                        onPlaylistClicked = { onPlaylistClicked(playlist) },
                        onDeleteClicked = { playlistManagementViewModel.deletePlaylist(playlis
                    )
                    HorizontalDivider()
                }
            }
        }

        // FAB is now part of the LibraryScreen scaffold, so it's removed from here
    }
}


@OptIn(ExperimentalMaterial3Api::class)
@Composable
private fun PlaylistItemRow(
    playlist: PlaylistEntity,
    onPlaylistClicked: () -> Unit,
    onDeleteClicked: () -> Unit
) {
    ListItem(
        headlineContent = { Text(playlist.name ?: "Untitled Playlist", maxLines = 1, overflow
        supportingContent = {
            playlist.description?.takeIf { it.isNotBlank() }?.let {
                Text(it, maxLines = 2, overflow = TextOverflow.Ellipsis, style = MaterialTheme
            }
        },
        leadingContent = {
            Icon(
                Icons.AutoMirrored.Filled.PlaylistPlay,
                contentDescription = "Playlist icon",
                modifier = Modifier.size(24.dp)
            )
        },
        trailingContent = {
            IconButton(onClick = onDeleteClicked) {
                Icon(Icons.Filled.Delete, contentDescription = stringResource(R.string.action_
```

```
            }
        },
        modifier = Modifier.clickable(onClick = onPlaylistClicked)
    )
}

// File: java\com\example\holodex\ui\screens\SettingsScreen.kt
// File: java/com/example/holodex/ui/screens/SettingsScreen.kt
package com.example.holodex.ui.screens


import android.Manifest
import android.content.pm.PackageManager
import android.os.Build
import android.widget.Toast
import androidx.activity.compose.rememberLauncherForActivityResult
import androidx.activity.result.contract.ActivityResultContracts
import androidx.compose.foundation.clickable
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.ColumnScope
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.rememberScrollState
import androidx.compose.foundation.selection.selectable
import androidx.compose.foundation.selection.selectableGroup
import androidx.compose.foundation.verticalScroll
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.automirrored.filled.ArrowBack
import androidx.compose.material.icons.automirrored.filled.Login
import androidx.compose.material.icons.filled.Clear
import androidx.compose.material.icons.filled.CloudSync
import androidx.compose.material.icons.filled.DocumentScanner
import androidx.compose.material.icons.filled.FolderOpen
import androidx.compose.material.icons.filled.Link
import androidx.compose.material3.Button
import androidx.compose.material3.ButtonDefaults
import androidx.compose.material3.CircularProgressIndicator
import androidx.compose.material3.ExperimentalMaterial3Api
import androidx.compose.material3.HorizontalDivider
import androidx.compose.material3.Icon
import androidx.compose.material3.IconButton
import androidx.compose.material3.ListItem
import androidx.compose.material3.ListItemDefaults
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.RadioButton
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Switch
import androidx.compose.material3.Text
import androidx.compose.material3.TextButton
```

```
import androidx.compose.material3.TopAppBar
import androidx.compose.runtime.Composable
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.runtime.collectAsState
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.platform.LocalUriHandler
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.semantics.Role
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.text.style.TextOverflow
import androidx.compose.ui.unit.dp
import androidx.core.content.ContextCompat
import androidx.hilt.navigation.compose.hiltViewModel
import androidx.lifecycle.compose.collectAsStateWithLifecycle
import androidx.media3.common.util.UnstableApi
import androidx.navigation.NavController
import com.example.holodex.BuildConfig
import com.example.holodex.R
import com.example.holodex.auth.AuthState
import com.example.holodex.auth.AuthViewModel
import com.example.holodex.ui.AppDestinations
import com.example.holodex.ui.composables.ApiKeyInputScreen
import com.example.holodex.viewmodel.AppPreferenceConstants
import com.example.holodex.viewmodel.ScanStatus
import com.example.holodex.viewmodel.SettingsViewModel
import com.example.holodex.viewmodel.ThemePreference
import timber.log.Timber

@OptIn(ExperimentalMaterial3Api::class)
@UnstableApi
@Composable
fun SettingsScreen(
    navController: NavController,
    onNavigateUp: () -> Unit,
    onApiKeySavedRestartNeeded: () -> Unit
) {
    val authViewModel: AuthViewModel = hiltViewModel()
    val settingsViewModel: SettingsViewModel = hiltViewModel()

    val context = LocalContext.current
    val authState by authViewModel.authState.collectAsStateWithLifecycle()
    val scanStatus by settingsViewModel.scanStatus.collectAsStateWithLifecycle()

    // --- START: Permission and Scan Launcher Logic ---
    val permissionLauncher = rememberLauncherForActivityResult(
        ActivityResultContracts.RequestPermission()
    ) { isGranted: Boolean ->
        if (isGranted) {
```

```
            Toast.makeText(context, "Permission granted. Starting scan...", Toast.LENGTH_SHORT
            settingsViewModel.runLegacyFileScan()
        } else {
            Toast.makeText(
                context,
                "Storage permission is required to find old downloads.",
                Toast.LENGTH_LONG
            ).show()
        }
    }
    val cacheClearStatus by settingsViewModel.cacheClearStatus.collectAsState()
    var isClearingCache by remember { mutableStateOf(false) }
    val currentThemePref by settingsViewModel.currentThemePreference.collectAsStateWithLifecyc

    val currentImageQuality by settingsViewModel.currentImageQuality.collectAsStateWithLifecyc
    val currentAudioQuality by settingsViewModel.currentAudioQuality.collectAsStateWithLifecyc
    val currentListLoadingConfig by settingsViewModel.currentListLoadingConfig.collectAsStateW
    val currentBufferingStrategy by settingsViewModel.currentBufferingStrategy.collectAsStateW
    val autoplayNextVideoEnabled by settingsViewModel.autoplayNextVideoEnabled.collectAsStateW
    val shuffleOnPlayStartEnabled by settingsViewModel.shuffleOnPlayStartEnabled.collectAsStat
    val downloadLocation by settingsViewModel.downloadLocation.collectAsStateWithLifecycle()
    val folderPickerLauncher = rememberLauncherForActivityResult(
        contract = ActivityResultContracts.OpenDocumentTree(),
        onResult = { uri ->
            if (uri != null) {
                Timber.d("Folder selected: $uri")
                settingsViewModel.saveDownloadLocation(uri)
            } else {
                Timber.d("Folder selection cancelled by user.")
            }
        }
    )
    var showRestartMessageForDataSettings by remember { mutableStateOf(false) }

    LaunchedEffect(cacheClearStatus) {
        cacheClearStatus?.let { status ->
            Toast.makeText(context, status, Toast.LENGTH_LONG).show()
            settingsViewModel.resetCacheClearStatus()
            isClearingCache = false
        }
    }

    LaunchedEffect(showRestartMessageForDataSettings) {
        if (showRestartMessageForDataSettings) {
            Toast.makeText(
                context,
                "List loading and buffering settings may require an app restart to apply.",
                Toast.LENGTH_LONG
            ).show()
            showRestartMessageForDataSettings = false
        }
    }

    LaunchedEffect(scanStatus) {
        when (val status = scanStatus) {
```

```
        is ScanStatus.Complete -> {
            val message = if (status.importedCount > 0) {
                "Successfully imported ${status.importedCount} file(s)!"
            } else {
                "Scan complete. No new files found."
            }
            Toast.makeText(context, message, Toast.LENGTH_LONG).show()
            settingsViewModel.resetScanStatus()
        }

        is ScanStatus.Error -> {
            Toast.makeText(context, status.message, Toast.LENGTH_LONG).show()
            settingsViewModel.resetScanStatus()
        }

        else -> { /* Idle or Scanning */
        }
    }
}

Scaffold(
    topBar = {
        TopAppBar(
            title = { Text(stringResource(R.string.settings_title)) },
            navigationIcon = {
                IconButton(onClick = onNavigateUp) {
                    Icon(
                        Icons.AutoMirrored.Filled.ArrowBack,
                        contentDescription = stringResource(R.string.action_back)
                    )
                }
            }
        )
    }
) { paddingValues ->
    Column(
        modifier = Modifier
            .padding(paddingValues)
            .fillMaxSize()
            .verticalScroll(rememberScrollState())
            .padding(horizontal = 16.dp),
        verticalArrangement = Arrangement.spacedBy(0.dp)
    ) {
        // --- API Key Section ---
        SettingsSectionTitle(stringResource(R.string.settings_section_api_key))
        ApiKeyInputScreen(
            settingsViewModel = settingsViewModel,
            onApiKeySavedSuccessfully = {
                onApiKeySavedRestartNeeded()
            },
            modifier = Modifier.padding(bottom = 16.dp)
        )

        HorizontalDivider()
        SettingsSectionTitle(stringResource(R.string.settings_section_account)) // <-- Add
```

```
when (val state = authState) {
    is AuthState.LoggedIn -> {
        // Show a "Logged In" status and a Logout button
        ListItem(
            headlineContent = { Text("Logged In") }, // Replace with user data lat
            supportingContent = { Text("Your data is being synchronized.") },
            leadingContent = {
                Icon(
                    Icons.Default.CloudSync,
                    contentDescription = null
                )
            }, // <-- Add this string
            trailingContent = {
                TextButton(onClick = { authViewModel.logout() }) {
                    Text(stringResource(R.string.action_logout)) // <-- Add this s
                }
            },
            colors = ListItemDefaults.colors(containerColor = Color.Transparent)
        )
        Button(
            onClick = { settingsViewModel.triggerManualSync() },
            modifier = Modifier
                .fillMaxWidth()
                .padding(horizontal = 16.dp, vertical = 8.dp)
        ) {
            Icon(
                Icons.Default.CloudSync,
                contentDescription = null,
                modifier = Modifier.size(ButtonDefaults.IconSize)
            )
            Spacer(modifier = Modifier.size(ButtonDefaults.IconSpacing))
            Text("Sync Now")
        }
    }


    is AuthState.LoggedOut, is AuthState.Error -> {
        // Show a Login button
        ListItem(
            headlineContent = { Text(stringResource(R.string.action_login)) }, //
            supportingContent = { Text(stringResource(R.string.settings_desc_login
            leadingContent = {
                Icon(
                    Icons.AutoMirrored.Filled.Login,
                    contentDescription = null
                )
            },
            modifier = Modifier.clickable {
                navController.navigate(AppDestinations.LOGIN_ROUTE)
            },
            colors = ListItemDefaults.colors(containerColor = Color.Transparent)
        )
        if (state is AuthState.Error) {
            Text(
```

```
                    text = "Last login attempt failed: ${state.message}",
                    color = MaterialTheme.colorScheme.error,
                    style = MaterialTheme.typography.bodySmall,
                    modifier = Modifier.padding(start = 16.dp, top = 4.dp)
                )
            }
        }

        is AuthState.InProgress -> {
            // Show a loading state if login is in progress
            ListItem(
                headlineContent = { Text("Logging in...") },
                leadingContent = { CircularProgressIndicator(modifier = Modifier.size(
                colors = ListItemDefaults.colors(containerColor = Color.Transparent)
            )
        }
    }
    HorizontalDivider()
    // --- Playback Settings Section (New Section Title) ---
    SettingsSectionTitle(stringResource(R.string.settings_section_playback)) // NEW st
    // NEW: Autoplay Next Video Toggle
    ListItem(
        headlineContent = { Text(stringResource(R.string.settings_label_autoplay_next_
        supportingContent = {
            Text(
                stringResource(R.string.settings_desc_autoplay_next_video),
                style = MaterialTheme.typography.bodySmall
            )
        }, // NEW string resource needed
        trailingContent = {
            Switch(
                checked = autoplayNextVideoEnabled,
                onCheckedChange = { isChecked ->
                    settingsViewModel.setAutoplayNextVideoEnabled(isChecked)
                }
            )
        },
        modifier = Modifier
            .fillMaxWidth()
            .clickable {
                // Make the whole row clickable to toggle the switch
                settingsViewModel.setAutoplayNextVideoEnabled(!autoplayNextVideoEnable
            },
        colors = ListItemDefaults.colors(containerColor = Color.Transparent)
    )
    ListItem(
        headlineContent = { Text(stringResource(R.string.settings_label_shuffle_on_pla
        supportingContent = {
            Text(
                stringResource(R.string.settings_desc_shuffle_on_play),
                style = MaterialTheme.typography.bodySmall
            )
        },
        trailingContent = {
            Switch(
```

```
                checked = shuffleOnPlayStartEnabled,
                onCheckedChange = { settingsViewModel.setShuffleOnPlayStartEnabled(it)
            )
        },
        modifier = Modifier
            .fillMaxWidth()
            .clickable { settingsViewModel.setShuffleOnPlayStartEnabled(!shuffleOnPlay
        colors = ListItemDefaults.colors(containerColor = Color.Transparent)
    )

    Spacer(modifier = Modifier.height(16.dp))
    HorizontalDivider()


    // --- Data & Performance Section ---
    SettingsSectionTitle(stringResource(R.string.settings_section_data_performance))

    // Download Location Preference

    ListItem(
        headlineContent = { Text("Import Legacy Downloads") },
        supportingContent = { Text("Scan the HolodexMusic folder for any downloads not
        leadingContent = {
            when (scanStatus) {
                is ScanStatus.Scanning -> CircularProgressIndicator(modifier = Modifie
                else -> Icon(Icons.Default.DocumentScanner, contentDescription = null)
            }
        },
        modifier = Modifier.clickable(enabled = scanStatus !is ScanStatus.Scanning) {
            val permission = if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU
                Manifest.permission.READ_MEDIA_AUDIO
            } else {
                Manifest.permission.READ_EXTERNAL_STORAGE
            }

            if (ContextCompat.checkSelfPermission(context, permission) == PackageManag
                settingsViewModel.runLegacyFileScan()
            } else {
                permissionLauncher.launch(permission)
            }
        },
        colors = ListItemDefaults.colors(containerColor = Color.Transparent)
    )

    PreferenceGroupTitle(stringResource(R.string.settings_label_download_location))
    ListItem(
        headlineContent = {
            Text(
                text = if (downloadLocation.isEmpty()) stringResource(R.string.setting
                style = MaterialTheme.typography.bodyLarge,
                maxLines = 1,
                overflow = TextOverflow.Ellipsis
            )
        },
        supportingContent = { Text(stringResource(R.string.settings_desc_download_loca
```

```
        leadingContent = { Icon(Icons.Default.FolderOpen, contentDescription = null) }
        modifier = Modifier.clickable {
            try {
                folderPickerLauncher.launch(null) // Launch the folder picker
            } catch (e: Exception) {
                Timber.e(e, "Failed to launch folder picker")
                Toast.makeText(context, "Could not open folder picker.", Toast.LENGTH_
                    .show()
            }
        },
        trailingContent = {
            if (downloadLocation.isNotEmpty()) {
                IconButton(onClick = { settingsViewModel.clearDownloadLocation() }) {
                    Icon(
                        Icons.Default.Clear,
                        contentDescription = stringResource(R.string.action_clear_loca
                    )
                }
            }
        },
        colors = ListItemDefaults.colors(containerColor = Color.Transparent)
    )

    // Image Quality
    PreferenceGroupTitle(stringResource(R.string.settings_label_image_quality))
    PreferenceRadioGroup {
        ImageQualityOptions.entries.forEach { quality -> // Using ImageQualityOptions
            PreferenceRadioButton(
                text = quality.displayName,
                selected = currentImageQuality == quality.key,
                onClick = {
                    settingsViewModel.setImageQualityPreference(quality.key)
                    // Image quality changes can sometimes benefit from restart, espec
                    // But usually dynamically applying size to ImageRequest is enough
                    // Only show restart message if explicitly needed for subtle cache
                    // For simplicity, we'll show it if user picks non-AUTO.
                    if (quality.key != AppPreferenceConstants.IMAGE_QUALITY_AUTO) {
                        showRestartMessageForDataSettings = true
                    }
                }
            )
        }
    }
    PreferenceDescription(stringResource(R.string.settings_desc_image_quality))

    // Audio Quality
    PreferenceGroupTitle(stringResource(R.string.settings_label_audio_quality))
    PreferenceRadioGroup {
        AudioQualityOptions.entries.forEach { quality -> // Using AudioQualityOptions
            PreferenceRadioButton(
                text = quality.displayName,
                selected = currentAudioQuality == quality.key,
                onClick = { settingsViewModel.setAudioQualityPreference(quality.key) }
            )
        }
```

```
        }
        PreferenceDescription(stringResource(R.string.settings_desc_audio_quality))

        // List Loading (Paging)
        PreferenceGroupTitle(stringResource(R.string.settings_label_list_loading_config))
        PreferenceRadioGroup {
            ListLoadingConfigOptions.entries.forEach { config -> // Using ListLoadingConfi
                PreferenceRadioButton(
                    text = config.displayName,
                    selected = currentListLoadingConfig == config.key,
                    onClick = {
                        settingsViewModel.setListLoadingConfigPreference(config.key)
                        showRestartMessageForDataSettings =
                            true // Paging changes require restart
                    }
                )
            }
        }
        PreferenceDescription(stringResource(R.string.settings_desc_list_loading_config))

        // Buffering Strategy (ExoPlayer)
        PreferenceGroupTitle(stringResource(R.string.settings_label_buffering_strategy)) /
        PreferenceRadioGroup {
            BufferingStrategyOptions.entries.forEach { strategy -> // Using BufferingStrat
                PreferenceRadioButton(
                    text = strategy.displayName,
                    selected = currentBufferingStrategy == strategy.key,
                    onClick = {
                        settingsViewModel.setBufferingStrategyPreference(strategy.key)
                        showRestartMessageForDataSettings =
                            true // ExoPlayer recreation requires restart
                    }
                )
            }
        }
        PreferenceDescription(stringResource(R.string.settings_desc_buffering_strategy)) /


        Spacer(modifier = Modifier.height(16.dp))
        HorizontalDivider()

        // --- Cache Management Section ---
        SettingsSectionTitle(stringResource(R.string.settings_section_cache))
        Row(
            verticalAlignment = Alignment.CenterVertically,
            modifier = Modifier
                .fillMaxWidth()
                .padding(vertical = 8.dp)
        ) {
            Button(
                onClick = {
                    isClearingCache = true
                    settingsViewModel.clearAllApplicationCaches()
                },
                enabled = !isClearingCache,
```

```
            colors = ButtonDefaults.buttonColors(
                containerColor = MaterialTheme.colorScheme.errorContainer,
                contentColor = MaterialTheme.colorScheme.onErrorContainer
            ),
            modifier = Modifier.weight(1f)
        ) {
            Text(stringResource(R.string.settings_button_clear_cache))
        }
        if (isClearingCache) {
            CircularProgressIndicator(
                modifier = Modifier
                    .size(24.dp)
                    .padding(start = 8.dp)
            )
        }
    }
    Text(
        stringResource(R.string.settings_desc_clear_cache),
        style = MaterialTheme.typography.bodySmall,
        modifier = Modifier.padding(top = 4.dp, bottom = 16.dp)
    )
    HorizontalDivider()

    // --- Theme Selection Section ---
    SettingsSectionTitle(stringResource(R.string.settings_section_theme))
    PreferenceRadioGroup {
        ThemePreferenceOptions.entries.forEach { themeOpt ->
            PreferenceRadioButton(
                text = themeOpt.displayName,
                selected = currentThemePref == themeOpt.key,
                onClick = { settingsViewModel.setThemePreference(themeOpt.key) }
            )
        }
    }
    Spacer(modifier = Modifier.height(16.dp))
    HorizontalDivider()

    // --- About Section ---
    SettingsSectionTitle(stringResource(R.string.settings_section_about))
    InfoRow(
        label = stringResource(R.string.settings_label_version),
        value = BuildConfig.VERSION_NAME
    )
    Spacer(Modifier.height(8.dp))
    Text(
        stringResource(R.string.settings_label_powered_by),
        style = MaterialTheme.typography.titleMedium,
        modifier = Modifier.padding(top = 8.dp, bottom = 4.dp)
    )
    LinkItem(
        text = stringResource(R.string.settings_link_holodex),
        url = "https://holodex.net",
    )
    LinkItem(
        text = stringResource(R.string.settings_link_newpipe),
```

```
                url = "https://newpipe.net/",
            )
            Spacer(modifier = Modifier.height(24.dp))
        }
    }
}

// --- Helper Composables (remain the same) ---
@Composable
private fun SettingsSectionTitle(title: String) {
    Text(
        text = title,
        style = MaterialTheme.typography.titleLarge,
        modifier = Modifier.padding(top = 24.dp, bottom = 8.dp)
    )
}

@Composable
private fun PreferenceGroupTitle(title: String) {
    Text(
        text = title,
        style = MaterialTheme.typography.titleMedium,
        modifier = Modifier.padding(top = 16.dp, bottom = 4.dp)
    )
}

@Composable
private fun PreferenceDescription(text: String) {
    Text(
        text = text,
        style = MaterialTheme.typography.bodySmall,
        color = MaterialTheme.colorScheme.onSurfaceVariant,
        modifier = Modifier.padding(start = 16.dp, end = 16.dp, bottom = 8.dp)
    )
}

@Composable
private fun PreferenceRadioGroup(
    content: @Composable ColumnScope.() -> Unit
) {
    Column(Modifier.selectableGroup()) {
        content()
    }
}

@Composable
private fun PreferenceRadioButton(
    text: String,
    selected: Boolean,
    onClick: () -> Unit,
    enabled: Boolean = true
) {
    ListItem(
        headlineContent = { Text(text, style = MaterialTheme.typography.bodyLarge) },
        leadingContent = {
```

```
            RadioButton(
                selected = selected,
                onClick = null,
                enabled = enabled
            )
        },
        modifier = Modifier
            .fillMaxWidth()
            .selectable(
                selected = selected,
                onClick = if (enabled) onClick else ({}),
                role = Role.RadioButton,
                enabled = enabled
            ),
        colors = ListItemDefaults.colors(containerColor = Color.Transparent)
    )
}

// --- Enums for Settings Options (Moved/Updated to use AppPreferenceConstants) ---
enum class ImageQualityOptions(val key: String, val displayName: String) {
    AUTO(AppPreferenceConstants.IMAGE_QUALITY_AUTO, "Auto (Recommended)"),
    MEDIUM(AppPreferenceConstants.IMAGE_QUALITY_MEDIUM, "Medium (Faster loading)"),
    LOW(AppPreferenceConstants.IMAGE_QUALITY_LOW, "Low (Data saver)")
}

enum class AudioQualityOptions(val key: String, val displayName: String) {
    BEST(AppPreferenceConstants.AUDIO_QUALITY_BEST, "Best Available"),
    STANDARD(AppPreferenceConstants.AUDIO_QUALITY_STANDARD, "Standard (~128kbps)"),
    SAVER(AppPreferenceConstants.AUDIO_QUALITY_SAVER, "Data Saver (~64kbps)")
}

enum class ListLoadingConfigOptions(val key: String, val displayName: String) {
    NORMAL(AppPreferenceConstants.LIST_LOADING_NORMAL, "Normal (Smooth scrolling)"),
    REDUCED(AppPreferenceConstants.LIST_LOADING_REDUCED, "Reduced (Less data, faster initial)"
    MINIMAL(AppPreferenceConstants.LIST_LOADING_MINIMAL, "Minimal (Data saver, slowest scroll)
}

enum class BufferingStrategyOptions(val key: String, val displayName: String) {
    AGGRESSIVE(AppPreferenceConstants.BUFFERING_STRATEGY_AGGRESSIVE, "Quick Start (Default)"),
    BALANCED(AppPreferenceConstants.BUFFERING_STRATEGY_BALANCED, "Balanced"),
    STABLE(AppPreferenceConstants.BUFFERING_STRATEGY_STABLE, "Stable Playback (More buffering)
}

enum class ThemePreferenceOptions(val key: String, val displayName: String) {
    LIGHT(ThemePreference.LIGHT, "Light"),
    DARK(ThemePreference.DARK, "Dark"),
    SYSTEM(ThemePreference.SYSTEM, "Follow System")
}

// --- InfoRow and LinkItem Composables (remain the same) ---
@Composable
private fun InfoRow(label: String, value: String) {
    Row(
        modifier = Modifier
            .fillMaxWidth()
```

```
            .padding(vertical = 4.dp),
        verticalAlignment = Alignment.CenterVertically
    ) {
        Text(
            text = label,
            style = MaterialTheme.typography.bodyLarge,
            fontWeight = FontWeight.Medium,
            modifier = Modifier.weight(0.4f)
        )
        Text(
            text = value,
            style = MaterialTheme.typography.bodyLarge,
            color = MaterialTheme.colorScheme.onSurfaceVariant,
            modifier = Modifier.weight(0.6f)
        )
    }
}

@OptIn(ExperimentalMaterial3Api::class)
@Composable
private fun LinkItem(text: String, url: String) {
    val uriHandler = LocalUriHandler.current
    val context = LocalContext.current // Ensure context is available for Toast

    ListItem(
        headlineContent = { Text(text, style = MaterialTheme.typography.bodyLarge) },
        trailingContent = {
            Icon(
                Icons.Filled.Link,
                contentDescription = stringResource(R.string.content_desc_external_link)
            )
        },
        modifier = Modifier.clickable {
            try {
                uriHandler.openUri(url)
            } catch (e: Exception) {
                Timber.e(e, "Failed to open URL: $url")
                Toast.makeText(context, "Could not open link.", Toast.LENGTH_SHORT).show()
            }
        },
        colors = ListItemDefaults.colors(containerColor = Color.Transparent)
    )
}

// File: java\com\example\holodex\ui\screens\VideoDetailsScreen.kt
// File: java/com/example/holodex/ui/screens/VideoDetailsScreen.kt
// File: java/com/example/holodex/ui/screens/VideoDetailsScreen.kt
@file:kotlin.OptIn(ExperimentalMaterial3Api::class, ExperimentalFoundationApi::class)

package com.example.holodex.ui.screens

import android.widget.Toast
import androidx.compose.foundation.BorderStroke
import androidx.compose.foundation.ExperimentalFoundationApi
import androidx.compose.foundation.layout.Arrangement
```

```
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.PaddingValues
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.aspectRatio
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.itemsIndexed
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.automirrored.filled.ArrowBack
import androidx.compose.material.icons.automirrored.filled.QueueMusic
import androidx.compose.material.icons.filled.Download
import androidx.compose.material.icons.filled.Favorite
import androidx.compose.material.icons.filled.FavoriteBorder
import androidx.compose.material.icons.filled.MusicNote
import androidx.compose.material.icons.filled.PlayArrow
import androidx.compose.material3.Button
import androidx.compose.material3.ButtonDefaults
import androidx.compose.material3.ExperimentalMaterial3Api
import androidx.compose.material3.HorizontalDivider
import androidx.compose.material3.Icon
import androidx.compose.material3.IconButton
import androidx.compose.material3.IconButtonDefaults
import androidx.compose.material3.LocalContentColor
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.OutlinedButton
import androidx.compose.material3.Scaffold
import androidx.compose.material3.SnackbarDuration
import androidx.compose.material3.SnackbarHost
import androidx.compose.material3.SnackbarHostState
import androidx.compose.material3.Text
import androidx.compose.material3.TopAppBar
import androidx.compose.material3.TopAppBarDefaults
import androidx.compose.runtime.Composable
import androidx.compose.runtime.CompositionLocalProvider
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.runtime.derivedStateOf
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableIntStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.layout.ContentScale
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.res.pluralStringResource
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.text.font.FontWeight
```

```
import androidx.compose.ui.text.style.TextOverflow
import androidx.compose.ui.unit.dp
import androidx.hilt.navigation.compose.hiltViewModel
import androidx.lifecycle.compose.collectAsStateWithLifecycle
import androidx.media3.common.util.UnstableApi
import androidx.navigation.NavController
import coil.compose.AsyncImage
import coil.request.ImageRequest
import com.example.holodex.R
import com.example.holodex.data.db.ExternalChannelEntity
import com.example.holodex.data.db.FavoriteChannelEntity
import com.example.holodex.data.model.HolodexVideoItem
import com.example.holodex.ui.composables.ErrorStateWithRetry
import com.example.holodex.ui.composables.LoadingState
import com.example.holodex.ui.composables.SimpleProcessedBackground
import com.example.holodex.ui.composables.UnifiedListItem
import com.example.holodex.util.DynamicTheme
import com.example.holodex.util.ThumbnailQuality
import com.example.holodex.util.findActivity
import com.example.holodex.util.getYouTubeThumbnailUrl
import com.example.holodex.viewmodel.FavoritesViewModel
import com.example.holodex.viewmodel.PlaylistManagementViewModel
import com.example.holodex.viewmodel.VideoDetailsViewModel
import com.example.holodex.viewmodel.VideoListViewModel
import org.orbitmvi.orbit.compose.collectAsState

@UnstableApi
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun VideoDetailsScreen(
    navController: NavController,
    onNavigateUp: () -> Unit,
) {
    val videoDetailsViewModel: VideoDetailsViewModel = hiltViewModel()
    // *** THE FIX: Get the activity-scoped ViewModel here ***
    val videoListViewModel: VideoListViewModel = hiltViewModel(findActivity())
    val favoritesViewModel: FavoritesViewModel = hiltViewModel()
    val playlistManagementViewModel: PlaylistManagementViewModel = hiltViewModel(findActivity(

    // Call the initialize function once when the screen is first composed.
    LaunchedEffect(Unit) {
        videoDetailsViewModel.initialize(videoListViewModel)
    }

    val videoDetails by videoDetailsViewModel.videoDetails.collectAsStateWithLifecycle()
    val favoritesState by favoritesViewModel.collectAsState()
    val isLoading by videoDetailsViewModel.isLoading.collectAsStateWithLifecycle()
    val error by videoDetailsViewModel.error.collectAsStateWithLifecycle()
    val transientMessage by videoDetailsViewModel.transientMessage.collectAsStateWithLifecycle
    val snackbarHostState = remember { SnackbarHostState() }
    val context = LocalContext.current
    val dynamicTheme by videoDetailsViewModel.dynamicTheme.collectAsStateWithLifecycle()

    val backgroundImageUrl by remember(videoDetails) {
        derivedStateOf { videoDetails?.id?.let { getYouTubeThumbnailUrl(it, ThumbnailQuality.M
```

```
    }

    LaunchedEffect(error) {
        error?.let {
            snackbarHostState.showSnackbar(message = it, duration = SnackbarDuration.Long)
            videoDetailsViewModel.clearError()
        }
    }
    LaunchedEffect(transientMessage) {
        transientMessage?.let {
            Toast.makeText(context, it, Toast.LENGTH_SHORT).show()
            videoDetailsViewModel.clearTransientMessage()
        }
    }

    Scaffold(
        snackbarHost = { SnackbarHost(snackbarHostState) },
        topBar = {
            TopAppBar(
                title = { Text(text = videoDetails?.title ?: "", maxLines = 1, overflow = Text
                navigationIcon = {
                    IconButton(onClick = onNavigateUp) {
                        Icon(Icons.AutoMirrored.Filled.ArrowBack, contentDescription = stringR
                    }
                },
                actions = {
                    videoDetails?.let { video ->
                        val isFavorited = favoritesState.favoriteChannels.any {
                            (it is FavoriteChannelEntity && it.id == video.channel.id) ||
                                    (it is ExternalChannelEntity && it.channelId == video.chan
                        }
                        IconButton(onClick = { favoritesViewModel.toggleFavoriteChannel(video)
                            Icon(
                                imageVector = if (isFavorited) Icons.Filled.Favorite else Icon
                                contentDescription = "Favorite Channel",
                                tint = if (isFavorited) dynamicTheme.primary else dynamicTheme
                            )
                        }
                    }
                },
                colors = TopAppBarDefaults.topAppBarColors(
                    containerColor = Color.Transparent,
                    titleContentColor = dynamicTheme.onPrimary,
                    navigationIconContentColor = dynamicTheme.onPrimary,
                    actionIconContentColor = dynamicTheme.onPrimary
                )
            )
        }
    ) { paddingValues ->
        Box(modifier = Modifier.fillMaxSize()) {
            SimpleProcessedBackground(artworkUri = backgroundImageUrl, dynamicColor = dynamicT
            CompositionLocalProvider(LocalContentColor provides dynamicTheme.onPrimary) {
                Box(modifier = Modifier.padding(paddingValues).fillMaxSize()) {
                    when {
                        isLoading && videoDetails == null -> LoadingState(message = stringReso
```

```
                        error != null && videoDetails == null -> ErrorStateWithRetry(
                            message = error!!,
                            onRetry = { videoDetailsViewModel.initialize(videoListViewModel) }
                        )
                        videoDetails != null -> {
                            VideoDetailsContent(
                                videoDetailsViewModel = videoDetailsViewModel,
                                navController = navController,
                                playlistManagementViewModel = playlistManagementViewModel,
                                dynamicTheme = dynamicTheme
                            )
                        }
                    }
                }
            }
        }
    }
}

@UnstableApi
@Composable
private fun VideoDetailsContent(
    navController: NavController,
    videoDetailsViewModel: VideoDetailsViewModel,
    playlistManagementViewModel: PlaylistManagementViewModel,
    dynamicTheme: DynamicTheme
) {
    val videoItem by videoDetailsViewModel.videoDetails.collectAsStateWithLifecycle()
    val songItems by videoDetailsViewModel.unifiedSongItems.collectAsStateWithLifecycle()

    val videoListViewModel: VideoListViewModel = hiltViewModel(findActivity())
    val favoritesViewModel: FavoritesViewModel = hiltViewModel()

    LazyColumn(modifier = Modifier.fillMaxSize(), contentPadding = PaddingValues(bottom = 80.d
        item {
            videoItem?.let {
                VideoDetailsHeader(
                    videoItem = it,
                    songCount = songItems.size,
                    onPlayAll = { videoDetailsViewModel.playAllSegments() },
                    onAddToQueue = { videoDetailsViewModel.addAllSegmentsToQueue() },
                    onDownloadAll = { videoDetailsViewModel.downloadAllSegments() },
                    dynamicTheme = dynamicTheme
                )
            }
            HorizontalDivider(color = LocalContentColor.current.copy(alpha = 0.2f))
        }

        if (songItems.isNotEmpty()) {
            itemsIndexed(
                songItems,
                key = { _, song -> song.stableId }
            ) { index, songItem ->
                UnifiedListItem(
                    item = songItem,
```

```
                    onItemClicked = { videoDetailsViewModel.playSegment(index) },
                    navController = navController,
                    videoListViewModel = videoListViewModel,
                    favoritesViewModel = favoritesViewModel,
                    playlistManagementViewModel = playlistManagementViewModel
                )
            }
        } else {
            // Display empty state only if the parent video has finished loading
            if (videoItem != null && !videoDetailsViewModel.isLoading.value) {
                item { EmptyStateMessage() }
            }
        }
    }
}

@Composable
private fun VideoDetailsHeader(
    videoItem: HolodexVideoItem,
    songCount: Int,
    onPlayAll: () -> Unit,
    onAddToQueue: () -> Unit,
    onDownloadAll: () -> Unit,
    dynamicTheme: DynamicTheme
) {
    val thumbnailUrls = remember(videoItem.id) {
        getYouTubeThumbnailUrl(videoItem.id, ThumbnailQuality.MAX)
    }
    var currentUrlIndex by remember(thumbnailUrls) { mutableIntStateOf(0) }

    Column(modifier = Modifier.padding(16.dp)) {
        AsyncImage(
            model = ImageRequest.Builder(LocalContext.current)
                .data(thumbnailUrls.getOrNull(currentUrlIndex))
                .placeholder(R.drawable.ic_placeholder_image)
                .error(R.drawable.ic_error_image)
                .crossfade(true).build(),
            onError = { if (currentUrlIndex < thumbnailUrls.lastIndex) currentUrlIndex++ },
            contentDescription = stringResource(R.string.video_thumbnail_description),
            contentScale = ContentScale.Crop,
            modifier = Modifier
                .fillMaxWidth()
                .aspectRatio(16f / 9f)
                .clip(MaterialTheme.shapes.medium)
        )
        Spacer(Modifier.height(16.dp))
        Text(videoItem.title, style = MaterialTheme.typography.headlineSmall, fontWeight = Fon
        Spacer(Modifier.height(4.dp))
        Text(videoItem.channel.name, style = MaterialTheme.typography.titleMedium)
        if (songCount > 0) {
            Spacer(Modifier.height(4.dp))
            Text(
                text = pluralStringResource(R.plurals.song_count, songCount, songCount),
                style = MaterialTheme.typography.bodyMedium
            )
```

```
        }
        Spacer(Modifier.height(16.dp))
        if (songCount > 0) {
            ActionButtons(
                onPlayAll = onPlayAll,
                onAddToQueue = onAddToQueue,
                onDownloadAll = onDownloadAll,
                dynamicTheme = dynamicTheme
            )
        }
        Spacer(Modifier.height(16.dp))
    }
}

@Composable
private fun ActionButtons(
    onPlayAll: () -> Unit,
    onAddToQueue: () -> Unit,
    onDownloadAll: () -> Unit,
    dynamicTheme: DynamicTheme
) {
    Row(modifier = Modifier.fillMaxWidth(), horizontalArrangement = Arrangement.spacedBy(8.dp)
        Button(
            onClick = onPlayAll,
            modifier = Modifier.weight(1f),
            colors = ButtonDefaults.buttonColors(
                containerColor = dynamicTheme.onPrimary,
                contentColor = dynamicTheme.primary
            )
        ) {
            Icon(Icons.Filled.PlayArrow, null, modifier = Modifier.size(ButtonDefaults.IconSiz
            Spacer(Modifier.size(ButtonDefaults.IconSpacing))
            Text(stringResource(R.string.action_play_all))
        }
        OutlinedButton(
            onClick = onAddToQueue,
            modifier = Modifier.weight(1f),
            colors = ButtonDefaults.outlinedButtonColors(contentColor = dynamicTheme.onPrimary
            border = BorderStroke(1.dp, dynamicTheme.onPrimary.copy(alpha = 0.5f))
        ) {
            Icon(Icons.AutoMirrored.Filled.QueueMusic, null, modifier = Modifier.size(ButtonDe
            Spacer(Modifier.size(ButtonDefaults.IconSpacing))
            Text(stringResource(R.string.action_add_to_queue_short))
        }
        IconButton(
            onClick = onDownloadAll,
            colors = IconButtonDefaults.iconButtonColors(contentColor = dynamicTheme.onPrimary
        ) {
            Icon(Icons.Filled.Download, stringResource(R.string.action_download_all))
        }
    }
}


@Composable
```

```
private fun EmptyStateMessage() {
    Box(
        modifier = Modifier
            .fillMaxWidth()
            .padding(32.dp), contentAlignment = Alignment.Center
    ) {
        Column(
            horizontalAlignment = Alignment.CenterHorizontally,
            verticalArrangement = Arrangement.spacedBy(8.dp)
        ) {
            Icon(
                Icons.Filled.MusicNote,
                contentDescription = null,
                tint = MaterialTheme.colorScheme.onSurfaceVariant,
                modifier = Modifier.size(48.dp)
            )
            Text(
                text = stringResource(R.string.no_song_segments_available),
                style = MaterialTheme.typography.bodyLarge,
                color = MaterialTheme.colorScheme.onSurfaceVariant
            )
        }
    }
}

// File: java\com\example\holodex\ui\screens\navigation\AppDestinations.kt
// File: java/com/example/holodex/ui/navigation/AppDestinations.kt
package com.example.holodex.ui.navigation

import com.example.holodex.viewmodel.FullListViewModel
import com.example.holodex.viewmodel.MusicCategoryType // FIX: Correct import
import com.example.holodex.viewmodel.PlaylistDetailsViewModel
import com.example.holodex.viewmodel.VideoDetailsViewModel
import java.net.URLEncoder
import java.nio.charset.StandardCharsets

object AppDestinations {
    const val HOME_ROUTE = "home"
    const val DISCOVERY_ROUTE = "discover"
    const val LIBRARY_ROUTE = "library"
    const val DOWNLOADS_ROUTE = "downloads"
    const val SETTINGS_ROUTE = "settings"
    const val LOGIN_ROUTE = "login"
    const val FOR_YOU_ROUTE = "for_you"

    const val VIDEO_DETAILS_ROUTE_TEMPLATE = "video_details/{${VideoDetailsViewModel.VIDEO_ID_
    fun videoDetailRoute(videoId: String) = "video_details/$videoId"

    const val FULL_LIST_VIEW_ROUTE_TEMPLATE =
        "full_list/{${FullListViewModel.CATEGORY_TYPE_ARG}}/{${FullListViewModel.ORG_ARG}}"

    // FIX: Use MusicCategoryType directly
    fun fullListViewRoute(category: MusicCategoryType, org: String): String {
        val encodedOrg = URLEncoder.encode(org, StandardCharsets.UTF_8.toString())
        return "full_list/${category.name}/$encodedOrg"
```

```
    }

    const val PLAYLIST_DETAILS_ROUTE_TEMPLATE =
        "playlist_details/{${PlaylistDetailsViewModel.PLAYLIST_ID_ARG}}"

    fun playlistDetailsRoute(playlistId: String): String {
        val encodedId = URLEncoder.encode(playlistId, StandardCharsets.UTF_8.toString())
        return "playlist_details/$encodedId"
    }
}

// File: java\com\example\holodex\ui\screens\navigation\BottomNavItem.kt
// --- FULL REPLACEMENT of the file content ---
package com.example.holodex.ui.screens.navigation

import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.Download
import androidx.compose.material.icons.filled.Explore
import androidx.compose.material.icons.filled.LibraryMusic
import androidx.compose.material.icons.filled.Search
import androidx.compose.ui.graphics.vector.ImageVector
import com.example.holodex.R
import com.example.holodex.ui.AppDestinations

sealed class BottomNavItem(
    val route: String,
    val titleResId: Int,
    val icon: ImageVector
) {
    object Discover : BottomNavItem(
        route = AppDestinations.DISCOVERY_ROUTE,
        titleResId = R.string.bottom_nav_discover, // Add string
        icon = Icons.Filled.Explore
    )
    object Browse : BottomNavItem(
        route = AppDestinations.HOME_ROUTE,
        titleResId = R.string.bottom_nav_browse,
        icon = Icons.Filled.Search
    )
    object Library : BottomNavItem(
        route = AppDestinations.LIBRARY_ROUTE,
        titleResId = R.string.bottom_nav_library,
        icon = Icons.Filled.LibraryMusic
    )
    object Downloads : BottomNavItem(
        route = AppDestinations.DOWNLOADS_ROUTE,
        titleResId = R.string.bottom_nav_downloads,
        icon = Icons.Filled.Download
    )
}

// File: java\com\example\holodex\ui\screens\navigation\HolodexNavHost.kt
package com.example.holodex.ui.navigation

import android.annotation.SuppressLint
```

```
import androidx.activity.ComponentActivity
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.Button
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
import androidx.hilt.navigation.compose.hiltViewModel
import androidx.lifecycle.compose.collectAsStateWithLifecycle
import androidx.navigation.NavHostController
import androidx.navigation.NavType
import androidx.navigation.compose.NavHost
import androidx.navigation.compose.composable
import androidx.navigation.navArgument
import com.example.holodex.auth.LoginScreen
import com.example.holodex.ui.screens.ChannelScreen
import com.example.holodex.ui.screens.DiscoveryScreen
import com.example.holodex.ui.screens.DownloadsScreen
import com.example.holodex.ui.screens.ExternalChannelScreen
import com.example.holodex.ui.screens.ForYouScreen
import com.example.holodex.ui.screens.FullListViewScreen
import com.example.holodex.ui.screens.HomeScreen
import com.example.holodex.ui.screens.LibraryScreen
import com.example.holodex.ui.screens.PlaylistDetailsScreen
import com.example.holodex.ui.screens.SettingsScreen
import com.example.holodex.ui.screens.VideoDetailsScreen
import com.example.holodex.viewmodel.ChannelDetailsViewModel
import com.example.holodex.viewmodel.FullListViewModel
import com.example.holodex.viewmodel.MusicCategoryType
import com.example.holodex.viewmodel.PlaylistDetailsViewModel
import com.example.holodex.viewmodel.PlaylistManagementViewModel
import com.example.holodex.viewmodel.SettingsViewModel
import com.example.holodex.viewmodel.VideoDetailsViewModel
import com.example.holodex.viewmodel.VideoListViewModel

@SuppressLint("UnstableApi") // For Media3
@Composable
fun HolodexNavHost(
    navController: NavHostController,
    videoListViewModel: VideoListViewModel,
    playlistManagementViewModel: PlaylistManagementViewModel,
    activity: ComponentActivity,
    modifier: Modifier = Modifier
) {
    NavHost(
        navController = navController,
        startDestination = AppDestinations.LIBRARY_ROUTE,
        modifier = modifier.fillMaxSize()
    ) {
        composable(AppDestinations.DISCOVERY_ROUTE) {
```

```
        DiscoveryScreen(navController = navController)
    }

    composable(AppDestinations.FOR_YOU_ROUTE) {
        ForYouScreen(navController = navController)
    }

    composable(AppDestinations.HOME_ROUTE) {
        val settingsViewModel: SettingsViewModel = hiltViewModel()
        val currentApiKey by settingsViewModel.currentApiKey.collectAsStateWithLifecycle()

        if (currentApiKey.isBlank()) {
            ApiKeyMissingContent(navController = navController)
        } else {
            HomeScreen(
                navController = navController,
                videoListViewModel = videoListViewModel,
                playlistManagementViewModel = playlistManagementViewModel
            )
        }
    }

    composable(AppDestinations.LIBRARY_ROUTE) {
        LibraryScreen(navController = navController, playlistManagementViewModel = playlis
    }

    composable(AppDestinations.DOWNLOADS_ROUTE) {
        DownloadsScreen(navController = navController, playlistManagementViewModel = playl
    }

    composable(AppDestinations.SETTINGS_ROUTE) {
        // We re-inject the Activity-Scoped VM here to ensure we refresh the list if API k
        val vListVm: VideoListViewModel = hiltViewModel(activity)
        SettingsScreen(
            navController = navController,
            onNavigateUp = { navController.popBackStack() },
            onApiKeySavedRestartNeeded = { vListVm.refreshCurrentListViaPull() }
        )
    }

    composable(AppDestinations.LOGIN_ROUTE) {
        LoginScreen(onLoginSuccess = { navController.popBackStack() })
    }

    // --- Details Screens (Parameterized) ---

    composable(
        route = "external_channel_details/{${ChannelDetailsViewModel.CHANNEL_ID_ARG}}",
        arguments = listOf(navArgument(ChannelDetailsViewModel.CHANNEL_ID_ARG) { type = Na
    ) {
        ExternalChannelScreen(navController = navController, onNavigateUp = { navControlle
    }

    composable(
        route = "channel_details/{${ChannelDetailsViewModel.CHANNEL_ID_ARG}}",
```

```
            arguments = listOf(navArgument(ChannelDetailsViewModel.CHANNEL_ID_ARG) { type = Na
        ) {
            ChannelScreen(navController = navController, onNavigateUp = { navController.popBac
        }

        composable(
            route = AppDestinations.FULL_LIST_VIEW_ROUTE_TEMPLATE,
            arguments = listOf(
                navArgument(FullListViewModel.CATEGORY_TYPE_ARG) { type = NavType.StringType }
                navArgument(FullListViewModel.ORG_ARG) { type = NavType.StringType }
            )
        ) { backStackEntry ->
            val categoryName = backStackEntry.arguments?.getString(FullListViewModel.CATEGORY_
            val category = try {
                MusicCategoryType.valueOf(categoryName)
            } catch (e: IllegalArgumentException) {
                MusicCategoryType.TRENDING
            }
            FullListViewScreen(navController = navController, categoryType = category)
        }

        composable(
            AppDestinations.PLAYLIST_DETAILS_ROUTE_TEMPLATE,
            arguments = listOf(navArgument(PlaylistDetailsViewModel.PLAYLIST_ID_ARG) { type =
        ) {
            PlaylistDetailsScreen(
                navController = navController,
                playlistManagementViewModel = playlistManagementViewModel,
                onNavigateUp = { navController.popBackStack() }
            )
        }

        composable(
            AppDestinations.VIDEO_DETAILS_ROUTE_TEMPLATE,
            arguments = listOf(navArgument(VideoDetailsViewModel.VIDEO_ID_ARG) { type = NavTyp
        ) {
            VideoDetailsScreen(navController = navController, onNavigateUp = { navController.p
        }
    }
}

@Composable
private fun ApiKeyMissingContent(navController: NavHostController) {
    Box(Modifier.fillMaxSize(), contentAlignment = Alignment.Center) {
        androidx.compose.foundation.layout.Column(horizontalAlignment = Alignment.CenterHorizo
            Text("API Key Required", style = MaterialTheme.typography.headlineSmall)
            Button(
                onClick = { navController.navigate(AppDestinations.SETTINGS_ROUTE) },
                modifier = Modifier.padding(top = 16.dp)
            ) {
                Text("Go to Settings")
            }
        }
    }
}
```

```
// File: java\com\example\holodex\ui\theme\Color.kt
package com.example.holodex.ui.theme

import androidx.compose.ui.graphics.Color

val md_theme_light_primary = Color(0xFF6750A4)
val md_theme_light_onPrimary = Color(0xFFFFFFFF)
val md_theme_light_primaryContainer = Color(0xFFEADDFF)
val md_theme_light_onPrimaryContainer = Color(0xFF21005D)
val md_theme_light_secondary = Color(0xFF625B71)
val md_theme_light_onSecondary = Color(0xFFFFFFFF)
val md_theme_light_secondaryContainer = Color(0xFFE8DEF8)
val md_theme_light_onSecondaryContainer = Color(0xFF1D192B)
val md_theme_light_tertiary = Color(0xFF7D5260)
val md_theme_light_onTertiary = Color(0xFFFFFFFF)
val md_theme_light_tertiaryContainer = Color(0xFFFFD8E4)
val md_theme_light_onTertiaryContainer = Color(0xFF31111D)
val md_theme_light_error = Color(0xFFB3261E)
val md_theme_light_onError = Color(0xFFFFFFFF)
val md_theme_light_errorContainer = Color(0xFFF9DEDC)
val md_theme_light_onErrorContainer = Color(0xFF410E0B)
val md_theme_light_background = Color(0xFFFFFBFE)
val md_theme_light_onBackground = Color(0xFF1C1B1F)
val md_theme_light_surface = Color(0xFFFFFBFE)
val md_theme_light_onSurface = Color(0xFF1C1B1F)
val md_theme_light_surfaceVariant = Color(0xFFE7E0EC)
val md_theme_light_onSurfaceVariant = Color(0xFF49454F)
val md_theme_light_outline = Color(0xFF79747E)
val md_theme_light_inverseOnSurface = Color(0xFFF4EFF4)
val md_theme_light_inverseSurface = Color(0xFF313033)
val md_theme_light_inversePrimary = Color(0xFFD0BCFF)
val md_theme_light_surfaceTint = Color(0xFF6750A4)
val md_theme_light_outlineVariant = Color(0xFFCAC4D0)
val md_theme_light_scrim = Color(0xFF000000)

val md_theme_dark_primary = Color(0xFFD0BCFF)
val md_theme_dark_onPrimary = Color(0xFF381E72)
val md_theme_dark_primaryContainer = Color(0xFF4F378B)
val md_theme_dark_onPrimaryContainer = Color(0xFFEADDFF)
val md_theme_dark_secondary = Color(0xFFCCC2DC)
val md_theme_dark_onSecondary = Color(0xFF332D41)
val md_theme_dark_secondaryContainer = Color(0xFF4A4458)
val md_theme_dark_onSecondaryContainer = Color(0xFFE8DEF8)
val md_theme_dark_tertiary = Color(0xFFEFB8C8)
val md_theme_dark_onTertiary = Color(0xFF492532)
val md_theme_dark_tertiaryContainer = Color(0xFF633B48)
val md_theme_dark_onTertiaryContainer = Color(0xFFFFD8E4)
val md_theme_dark_error = Color(0xFFF2B8B5)
val md_theme_dark_onError = Color(0xFF601410)
val md_theme_dark_errorContainer = Color(0xFF8C1D18)
val md_theme_dark_onErrorContainer = Color(0xFFF9DEDC)
val md_theme_dark_background = Color(0xFF1C1B1F)
val md_theme_dark_onBackground = Color(0xFFE6E1E5)
val md_theme_dark_surface = Color(0xFF1C1B1F)
```

```
val md_theme_dark_onSurface = Color(0xFFE6E1E5)
val md_theme_dark_surfaceVariant = Color(0xFF49454F)
val md_theme_dark_onSurfaceVariant = Color(0xFFCAC4D0)
val md_theme_dark_outline = Color(0xFF938F99)
val md_theme_dark_inverseOnSurface = Color(0xFF1C1B1F)
val md_theme_dark_inverseSurface = Color(0xFFE6E1E5)
val md_theme_dark_inversePrimary = Color(0xFF6750A4)
val md_theme_dark_surfaceTint = Color(0xFFD0BCFF)
val md_theme_dark_outlineVariant = Color(0xFF49454F)
val md_theme_dark_scrim = Color(0xFF000000)

// File: java\com\example\holodex\ui\theme\Shape.kt
package com.example.holodex.ui.theme

import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material3.Shapes
import androidx.compose.ui.unit.dp

val Shapes = Shapes(
    small = RoundedCornerShape(4.dp),
    medium = RoundedCornerShape(8.dp),
    large = RoundedCornerShape(12.dp)
)

// File: java\com\example\holodex\ui\theme\Theme.kt
// File: java/com/example/holodex/ui/theme/Theme.kt
package com.example.holodex.ui.theme

import android.os.Build
import androidx.compose.foundation.isSystemInDarkTheme
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.darkColorScheme
import androidx.compose.material3.dynamicDarkColorScheme
import androidx.compose.material3.dynamicLightColorScheme
import androidx.compose.material3.lightColorScheme
import androidx.compose.runtime.Composable
import androidx.compose.runtime.SideEffect
import androidx.compose.runtime.collectAsState
import androidx.compose.runtime.getValue
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.platform.LocalContext
import androidx.hilt.navigation.compose.hiltViewModel
import com.example.holodex.viewmodel.SettingsViewModel
import com.example.holodex.viewmodel.ThemePreference
import com.google.accompanist.systemuicontroller.rememberSystemUiController

// Your color definitions from Color.kt are implicitly available if in the same package,
// or import them if Color.kt is in a different sub-package.
// Assuming they are in the same package 'com.example.holodex.ui.theme'

private val AppDarkColorScheme = darkColorScheme(
    primary = md_theme_dark_primary,
    onPrimary = md_theme_dark_onPrimary,
    primaryContainer = md_theme_dark_primaryContainer,
    onPrimaryContainer = md_theme_dark_onPrimaryContainer,
```

```
    secondary = md_theme_dark_secondary,
    onSecondary = md_theme_dark_onSecondary,
    secondaryContainer = md_theme_dark_secondaryContainer,
    onSecondaryContainer = md_theme_dark_onSecondaryContainer,
    tertiary = md_theme_dark_tertiary,
    onTertiary = md_theme_dark_onTertiary,
    tertiaryContainer = md_theme_dark_tertiaryContainer,
    onTertiaryContainer = md_theme_dark_onTertiaryContainer,
    error = md_theme_dark_error,
    onError = md_theme_dark_onError,
    errorContainer = md_theme_dark_errorContainer,
    onErrorContainer = md_theme_dark_onErrorContainer,
    background = md_theme_dark_background,
    onBackground = md_theme_dark_onBackground,
    surface = md_theme_dark_surface, // Crucial for NavigationBar color
    onSurface = md_theme_dark_onSurface,
    surfaceVariant = md_theme_dark_surfaceVariant,
    onSurfaceVariant = md_theme_dark_onSurfaceVariant,
    outline = md_theme_dark_outline,
    inverseOnSurface = md_theme_dark_inverseOnSurface,
    inverseSurface = md_theme_dark_inverseSurface,
    inversePrimary = md_theme_dark_inversePrimary,
    surfaceTint = md_theme_dark_surfaceTint,
    outlineVariant = md_theme_dark_outlineVariant,
    scrim = md_theme_dark_scrim,
)

private val AppLightColorScheme = lightColorScheme(
    primary = md_theme_light_primary,
    onPrimary = md_theme_light_onPrimary,
    primaryContainer = md_theme_light_primaryContainer,
    onPrimaryContainer = md_theme_light_onPrimaryContainer,
    secondary = md_theme_light_secondary,
    onSecondary = md_theme_light_onSecondary,
    secondaryContainer = md_theme_light_secondaryContainer,
    onSecondaryContainer = md_theme_light_onSecondaryContainer,
    tertiary = md_theme_light_tertiary,
    onTertiary = md_theme_light_onTertiary,
    tertiaryContainer = md_theme_light_tertiaryContainer,
    onTertiaryContainer = md_theme_light_onTertiaryContainer,
    error = md_theme_light_error,
    onError = md_theme_light_onError,
    errorContainer = md_theme_light_errorContainer,
    onErrorContainer = md_theme_light_onErrorContainer,
    background = md_theme_light_background,
    onBackground = md_theme_light_onBackground,
    surface = md_theme_light_surface, // Crucial for NavigationBar color
    onSurface = md_theme_light_onSurface,
    surfaceVariant = md_theme_light_surfaceVariant,
    onSurfaceVariant = md_theme_light_onSurfaceVariant,
    outline = md_theme_light_outline,
    inverseOnSurface = md_theme_light_inverseOnSurface,
    inverseSurface = md_theme_light_inverseSurface,
    inversePrimary = md_theme_light_inversePrimary,
    surfaceTint = md_theme_light_surfaceTint,
```

```
    outlineVariant = md_theme_light_outlineVariant,
    scrim = md_theme_light_scrim,
)

@Composable
fun HolodexMusicTheme(
    settingsViewModel: SettingsViewModel = hiltViewModel(),
    dynamicColor: Boolean = true, // Monet support (Android 12+)
    content: @Composable () -> Unit
) {
    val themePreference by settingsViewModel.currentThemePreference.collectAsState()

    val useDarkTheme = when (themePreference) {
        ThemePreference.LIGHT -> false
        ThemePreference.DARK -> true
        else -> isSystemInDarkTheme() // ThemePreference.SYSTEM or default
    }

    val colorScheme = when {
        dynamicColor && Build.VERSION.SDK_INT >= Build.VERSION_CODES.S -> {
            val context = LocalContext.current
            if (useDarkTheme) dynamicDarkColorScheme(context) else dynamicLightColorScheme(con
        }
        // --- Use your AppColorScheme variables now ---
        useDarkTheme -> AppDarkColorScheme
        else -> AppLightColorScheme
    }

    // System UI Controller for status/nav bar colors (edge-to-edge)
    // This SideEffect should be here, within HolodexMusicTheme, so it reacts to colorScheme c
    // If it's in MainActivity -> HolodexApp, it might not recompose when only the theme chang
    val systemUiController = rememberSystemUiController()
    SideEffect {
        systemUiController.setStatusBarColor(
            color = Color.Transparent,
            darkIcons = !useDarkTheme // Dark icons on light status bar, light icons on dark s
        )
        systemUiController.setNavigationBarColor(
            color = Color.Transparent, // System nav bar transparent for edge-to-edge
            darkIcons = !useDarkTheme,
            navigationBarContrastEnforced = false // Allows full transparency
        )
    }

    MaterialTheme(
        colorScheme = colorScheme,
        typography = Typography, // Assuming Typography is defined in Type.kt
        shapes = Shapes,         // Assuming Shapes is defined in Shape.kt
        content = content
    )
}

// File: java\com\example\holodex\ui\theme\Type.kt
package com.example.holodex.ui.theme
```

```
import androidx.compose.material3.Typography
import androidx.compose.ui.text.TextStyle
import androidx.compose.ui.text.font.FontFamily
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.unit.sp

// Replace with your own font families if desired
val Typography = Typography(
    bodyLarge = TextStyle(
        fontFamily = FontFamily.Default,
        fontWeight = FontWeight.Normal,
        fontSize = 16.sp,
        lineHeight = 24.sp,
        letterSpacing = 0.5.sp
    ),
    titleLarge = TextStyle(
        fontFamily = FontFamily.Default,
        fontWeight = FontWeight.Normal,
        fontSize = 22.sp,
        lineHeight = 28.sp,
        letterSpacing = 0.sp
    ),
    labelSmall = TextStyle(
        fontFamily = FontFamily.Default,
        fontWeight = FontWeight.Medium,
        fontSize = 11.sp,
        lineHeight = 16.sp,
        letterSpacing = 0.5.sp
    )
    // Add other text styles as needed
)

// File: java\com\example\holodex\util\ArtworkResolver.kt
package com.example.holodex.util

import com.example.holodex.data.model.discovery.PlaylistStub
import timber.log.Timber
import java.util.regex.Pattern

/**
 * A utility object to resolve the best possible artwork URL for different data models.
 */
object ArtworkResolver {

    // Regex to extract channel ID from playlist IDs like ":dailyrandom[ch=...]" or ":artist[c
    private val CHANNEL_ID_PATTERN: Pattern = Pattern.compile("ch=([a-zA-Z0-9_-]{24})")

    // --- START OF IMPLEMENTATION ---
    /**
     * Constructs the standard URL for a channel's photo based on its ID.
     * This is used as a fallback when the API does not provide a direct photo URL.
     *
     * @param channelId The unique ID of the channel.
     * @return The fully-formed URL to the channel's 200px profile picture.
     */
```

```
    fun getChannelPhotoUrl(channelId: String): String {
        return "https://holodex.net/statics/channelImg/$channelId/200.png"
    }
    // --- END OF IMPLEMENTATION ---

    /**
     * The main function to resolve playlist artwork. It follows the fallback logic
     * discovered from the Musicdex frontend.
     *
     * @param playlist The PlaylistStub object from the API.
     * @return The best available URL string for the playlist's artwork, or null if none can b
     */
    fun getPlaylistArtworkUrl(playlist: PlaylistStub): String? {
        Timber.d("Resolving artwork for playlist: ${playlist.title} (Type: ${playlist.type})")

        // Method 1: Use the pre-defined channel image for specific types
        if (playlist.type.startsWith(":dailyrandom") || playlist.type.startsWith(":artist")) {
            val matcher = CHANNEL_ID_PATTERN.matcher(playlist.id)
            if (matcher.find()) {
                val channelId = matcher.group(1)
                if (!channelId.isNullOrBlank()) {
                    Timber.d("PlaylistArtwork: Using Method 1 (Channel ID from playlist ID)")
                    return "https://holodex.net/statics/channelImg/$channelId/200.png"
                }
            }
        }

        // Method 2: Use the art_context field
        val artContext = playlist.artContext
        if (artContext != null) {
            // Rule from Musicdex: If it seems channel-focused, use the channel image.
            val isChannelFocused = (artContext.channels?.size ?: 0) < 3 && (artContext.videos?
            if (isChannelFocused && !artContext.channels.isNullOrEmpty()) {
                Timber.d("PlaylistArtwork: Using Method 2 (Channel-focused art_context)")
                return "https://holodex.net/statics/channelImg/${artContext.channels.first()}/
            }
            // Otherwise, assume it's video-focused.
            if (!artContext.videos.isNullOrEmpty()) {
                Timber.d("PlaylistArtwork: Using Method 2 (Video-focused art_context)")
                return getYouTubeThumbnailUrl(artContext.videos.first(), ThumbnailQuality.HIGH
            }
            // Fallback within art_context to channel photo if no videos
            if (!artContext.channelPhotoUrl.isNullOrBlank()) {
                Timber.d("PlaylistArtwork: Using Method 2 (Fallback channel photo from art_con
                return artContext.channelPhotoUrl
            }
        }

        Timber.w("PlaylistArtwork: No suitable URL found for playlist '${playlist.title}' usin
        return null
    }
}

// File: java\com\example\holodex\util\ComposableUtils.kt
package com.example.holodex.util
```

```
import android.content.ContextWrapper
import androidx.activity.ComponentActivity
import androidx.compose.runtime.Composable
import androidx.compose.runtime.remember
import androidx.compose.ui.platform.LocalContext

@Composable
fun findActivity(): ComponentActivity {
    val context = LocalContext.current
    return remember(context) {
        var a = context
        while (a is ContextWrapper) {
            if (a is ComponentActivity) {
                return@remember a
            }
            a = a.baseContext
        }
        // This should not happen in a normal app setup
        error("Could not find activity context")
    }
}

// File: java\com\example\holodex\util\Extract_util.kt
// File: java/com/example/holodex/util/Extract_util.kt
package com.example.holodex.util

import timber.log.Timber

// Made into a top-level function as it was called that way
// Alternatively, make Extract_util an object and call Extract_util.extractVideoIdFromQuery
fun extractVideoIdFromQuery(query: String): String? {
    val trimmedQuery = query.trim()
    val videoIdRegex = Regex("^[a-zA-Z0-9_-]{11}$")
    if (videoIdRegex.matches(trimmedQuery)) {
        Timber.Forest.d("extractVideoIdFromQuery: Matched direct video ID: $trimmedQuery")
        return trimmedQuery
    }

    // Regex for Holodex Music URL (music.holodex.net/video/ID or holodex.net/watch/ID)
    // Allows for optional trailing slashes or query params after ID
    val holodexUrlRegex = Regex("""^https://(music\.)?holodex\.net/(video|watch)/([a-zA-Z0-9_-
    var matcher = holodexUrlRegex.find(trimmedQuery)
    if (matcher != null) {
        val videoId = matcher.groupValues[3] // Group 3 is the ID
        Timber.Forest.d("extractVideoIdFromQuery: Matched Holodex URL, extracted ID: $videoId"
        return videoId
    }

    // Regex for YouTube URLs (youtube.com/watch?v=ID or youtu.be/ID)
    // More comprehensive regex to catch various YouTube URL formats
    val youtubeUrlRegex = Regex("""(?:youtube\.com/(?:[^/]+/.+/|(?:v|e(?:mbed)?)/|.*[?&]v=)|yo
    matcher = youtubeUrlRegex.find(trimmedQuery)
    if (matcher != null) {
        val videoId = matcher.groupValues[1] // Group 1 is the ID
```

```
        Timber.Forest.d("extractVideoIdFromQuery: Matched YouTube URL, extracted ID: $videoId")
        return videoId
    }

    Timber.Forest.d("extractVideoIdFromQuery: No video ID extracted from query: $trimmedQuery")
    return null
}


// File: java\com\example\holodex\util\ImageUtils.kt
// File: java/com/example/holodex/util/ImageUtils.kt
package com.example.holodex.util // Or your preferred util package

import com.example.holodex.playback.domain.model.PlaybackItem
import com.example.holodex.viewmodel.AppPreferenceConstants
import timber.log.Timber

// Regex to find the size part of an iTunes/mzstatic image URL
private val ITUNES_ARTWORK_SIZE_REGEX = Regex("""/(\d+x\d+)(bb)?\.jpg$""")

// Default sizes remain, but will be overridden by preference
private const val DEFAULT_HIGH_RES_SIZE = "600x600"
private const val MEDIUM_RES_SIZE = "300x300"
private const val LOW_RES_SIZE = "150x150"

enum class ThumbnailQuality {
    LOW,    // For tiny notifications or widgets (default.jpg - 120x90)
    MEDIUM, // For list items (mqdefault.jpg - 320x180)
    HIGH,   // For larger cards or mini-player (hqdefault.jpg - 480x360)
    MAX     // For full-screen displays (maxresdefault.jpg - 1280x720)
}

/**
 * NEW: Generates a prioritized list of YouTube thumbnail URLs for a given video ID.
 * Coil will attempt to load them in the order provided.
 */
fun getYouTubeThumbnailUrl(videoId: String, quality: ThumbnailQuality): List<String> {
    val baseUrl = "https://i.ytimg.com/vi/$videoId"
    return when (quality) {
        ThumbnailQuality.MAX -> listOf(
            "$baseUrl/maxresdefault.jpg",
            "$baseUrl/sddefault.jpg",
            "$baseUrl/hqdefault.jpg"
        )
        ThumbnailQuality.HIGH -> listOf(
            "$baseUrl/hqdefault.jpg",
            "$baseUrl/mqdefault.jpg",
            "$baseUrl/sddefault.jpg" // sddefault is often better than mqdefault
        )
        ThumbnailQuality.MEDIUM -> listOf(
            "$baseUrl/mqdefault.jpg",
            "$baseUrl/default.jpg"
        )
        ThumbnailQuality.LOW -> listOf(
            "$baseUrl/default.jpg"
```

```
        )
    }
}


// Updated function to take imageQualityKey as a parameter
fun getHighResArtworkUrl(
    originalUrl: String?,
    imageQualityKey: String = AppPreferenceConstants.IMAGE_QUALITY_AUTO, // Default to AUTO
    preferredSizeOverride: String? = null // Allows specific components (like FullPlayer) to s
): String? {
    if (originalUrl.isNullOrBlank()) {
        return null
    }

    if (!originalUrl.contains("mzstatic.com")) {
        return originalUrl // Return original if not an mzstatic URL
    }

    val targetSize = preferredSizeOverride ?: when (imageQualityKey) {
        AppPreferenceConstants.IMAGE_QUALITY_LOW -> LOW_RES_SIZE
        AppPreferenceConstants.IMAGE_QUALITY_MEDIUM -> MEDIUM_RES_SIZE
        AppPreferenceConstants.IMAGE_QUALITY_AUTO -> DEFAULT_HIGH_RES_SIZE // "Auto" here mean
        else -> DEFAULT_HIGH_RES_SIZE // Fallback
    }

    return ITUNES_ARTWORK_SIZE_REGEX.replace(originalUrl) { matchResult ->
        val bbSuffix = matchResult.groupValues.getOrNull(2) ?: ""
        Timber.d("getHighResArtworkUrl: URL: '$originalUrl', Quality: '$imageQualityKey', Targ
        "/${targetSize}${bbSuffix}.jpg"
    }.ifEmpty { originalUrl }
}
/**
 * Generates a prioritized, context-aware list of artwork URLs for a given PlaybackItem.
 * Coil will attempt to load from this list in order, using the first one that succeeds.
 *
 * @param item The PlaybackItem to get artwork for.
 * @param quality The desired quality for the *fallback* YouTube thumbnail. This is key
 * for requesting a high-res image for the player and a medium-res one for list items.
 * @return A list of URL strings in order of priority.
 */
fun generateArtworkUrlList(item: PlaybackItem?, quality: ThumbnailQuality): List<String> {
    if (item == null) return emptyList()

    val urls = mutableListOf<String>()

    // PRIORITY 1: The song's specific artwork URI from mzstatic.
    // We will use getHighResArtworkUrl to ensure we get a decently sized version.
    if (!item.artworkUri.isNullOrBlank() && item.artworkUri.contains("mzstatic.com")) {
        // For song-specific art, we usually want a high quality version regardless of context
        // We can use the existing utility for this.
        val highResSongArt = getHighResArtworkUrl(item.artworkUri, AppPreferenceConstants.IMAG
        if(highResSongArt != null) {
            urls.add(highResSongArt)
        }
```

```
    }

    // PRIORITY 2 (FALLBACK): YouTube thumbnails for the parent video, at the requested qualit
    urls.addAll(getYouTubeThumbnailUrl(item.videoId, quality))

    // As a final fallback, you could add the channel photo, but for now, we'll stick to
    // the user's request: song art -> video thumbnail.
    // if (!item.artworkUri.isNullOrBlank()) { urls.add(item.artworkUri) } // Fallback to chan

    return urls.distinct()
}

// File: java\com\example\holodex\util\PaletteExtractor.kt
package com.example.holodex.util

import android.content.Context
import android.graphics.Bitmap
import android.graphics.drawable.BitmapDrawable
import androidx.annotation.ColorInt
import androidx.collection.LruCache
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.graphics.toArgb
import androidx.core.graphics.ColorUtils
import androidx.palette.graphics.Palette
import coil.imageLoader
import coil.request.ImageRequest
import coil.request.SuccessResult
import coil.size.Size
import dagger.hilt.android.qualifiers.ApplicationContext
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.withContext
import timber.log.Timber
import javax.inject.Inject

// Data class to hold the extracted theme colors - keeping original structure
data class DynamicTheme(
    val primary: Color,
    val onPrimary: Color,
) {
    companion object {
        fun default(defaultPrimary: Color, defaultOnPrimary: Color) = DynamicTheme(
            primary = defaultPrimary,
            onPrimary = defaultOnPrimary
        )
    }
}

/**
 * Extracts a color palette from a given image URL with enhanced blur processing and in-memory
 */
class PaletteExtractor @Inject constructor(
    @ApplicationContext private val context: Context
) {
    private val cache = LruCache<String, DynamicTheme>(20) // Keep original cache size
```

```
suspend fun extractThemeFromUrl(
    imageUrl: String?,
    defaultTheme: DynamicTheme
): DynamicTheme = withContext(Dispatchers.Default) {
    if (imageUrl.isNullOrBlank()) return@withContext defaultTheme

    // Return from cache if available
    cache.get(imageUrl)?.let {
        Timber.d("PaletteExtractor: Cache HIT for $imageUrl")
        return@withContext it
    }

    Timber.d("PaletteExtractor: Cache MISS for $imageUrl. Processing.")
    try {
        val request = ImageRequest.Builder(context)
            .data(imageUrl)
            .size(Size(256, 256)) // Slightly larger for better blur quality while keeping
            .allowHardware(false) // Palette requires software bitmaps
            .memoryCacheKey("${imageUrl}_palette_enhanced")
            .build()

        val result = (context.imageLoader.execute(request) as? SuccessResult)?.drawable
        val bitmap = (result as? BitmapDrawable)?.bitmap ?: return@withContext defaultThem

        // Create a more beautiful blurred version of the bitmap for better color extracti
        val enhancedBitmap = createEnhancedBitmap(bitmap)

        val palette = Palette.from(enhancedBitmap).generate()

        val swatch = palette.vibrantSwatch
            ?: palette.lightVibrantSwatch
            ?: palette.darkVibrantSwatch
            ?: palette.dominantSwatch
            ?: palette.mutedSwatch
            ?: palette.lightMutedSwatch
            ?: palette.darkMutedSwatch
            ?: return@withContext defaultTheme

        val primaryColor = Color(swatch.rgb)
        val onPrimaryColor = Color(getBestTextColorForBackground(swatch.rgb, Color.White.t

        val newTheme = DynamicTheme(primary = primaryColor, onPrimary = onPrimaryColor)
        cache.put(imageUrl, newTheme) // Store in cache
        return@withContext newTheme

    } catch (e: Exception) {
        Timber.e(e, "Failed to extract palette from URL: $imageUrl")
        return@withContext defaultTheme
    }
}

/**
 * Creates an enhanced version of the bitmap with better color saturation and slight blur
 * for more beautiful color extraction without affecting the original API
 */
```

```
private fun createEnhancedBitmap(originalBitmap: Bitmap): Bitmap {
    return try {
        val config = originalBitmap.config ?: Bitmap.Config.ARGB_8888
        val enhancedBitmap = originalBitmap.copy(config, false)

        // Apply a gentle blur to smooth out harsh details and create more cohesive colors
        val blurredBitmap = applyGaussianBlur(enhancedBitmap, 3f)

        // Enhance color saturation for more vibrant palette extraction
        enhanceSaturation(blurredBitmap, 1.2f)
    } catch (e: Exception) {
        Timber.w(e, "Failed to enhance bitmap, using original")
        originalBitmap
    }
}

/**
 * Apply Gaussian blur using a modern, efficient algorithm
 */
private fun applyGaussianBlur(bitmap: Bitmap, radius: Float): Bitmap {
    if (radius <= 0) return bitmap

    val config = bitmap.config ?: Bitmap.Config.ARGB_8888
    val blurred = bitmap.copy(config, true)

    val width = blurred.width
    val height = blurred.height
    val pixels = IntArray(width * height)
    blurred.getPixels(pixels, 0, width, 0, 0, width, height)

    // Apply horizontal blur
    blurPixels(pixels, width, height, radius.toInt(), true)
    // Apply vertical blur
    blurPixels(pixels, width, height, radius.toInt(), false)

    blurred.setPixels(pixels, 0, width, 0, 0, width, height)
    return blurred
}

/**
 * Efficient box blur implementation for horizontal and vertical passes
 */
private fun blurPixels(pixels: IntArray, width: Int, height: Int, radius: Int, horizontal:
    val blur = IntArray(pixels.size)
    val kernel = createGaussianKernel(radius)
    val kernelSize = kernel.size
    val kernelRadius = kernelSize / 2

    for (y in 0 until height) {
        for (x in 0 until width) {
            var r = 0f
            var g = 0f
            var b = 0f
            var a = 0f
```

```
            for (k in 0 until kernelSize) {
                val weight = kernel[k]
                val sampleX = if (horizontal) {
                    (x + k - kernelRadius).coerceIn(0, width - 1)
                } else x
                val sampleY = if (horizontal) y else {
                    (y + k - kernelRadius).coerceIn(0, height - 1)
                }

                val pixel = pixels[sampleY * width + sampleX]
                r += ((pixel shr 16) and 0xFF) * weight
                g += ((pixel shr 8) and 0xFF) * weight
                b += (pixel and 0xFF) * weight
                a += ((pixel shr 24) and 0xFF) * weight
            }

            val blurredPixel = (a.toInt() shl 24) or
                    (r.toInt() shl 16) or
                    (g.toInt() shl 8) or
                    b.toInt()
            blur[y * width + x] = blurredPixel
        }
    }

    System.arraycopy(blur, 0, pixels, 0, pixels.size)
}

/**
 * Create a Gaussian kernel for blur
 */
private fun createGaussianKernel(radius: Int): FloatArray {
    val size = radius * 2 + 1
    val kernel = FloatArray(size)
    val sigma = radius / 3f
    var sum = 0f

    for (i in kernel.indices) {
        val x = i - radius
        kernel[i] = kotlin.math.exp(-(x * x) / (2 * sigma * sigma))
        sum += kernel[i]
    }

    // Normalize
    for (i in kernel.indices) {
        kernel[i] /= sum
    }

    return kernel
}

/**
 * Enhance color saturation for more vibrant palette extraction
 */
private fun enhanceSaturation(bitmap: Bitmap, factor: Float): Bitmap {
    val width = bitmap.width
```

```
        val height = bitmap.height
        val pixels = IntArray(width * height)
        bitmap.getPixels(pixels, 0, width, 0, 0, width, height)

        for (i in pixels.indices) {
            val pixel = pixels[i]
            val hsv = FloatArray(3)
            val r = (pixel shr 16) and 0xFF
            val g = (pixel shr 8) and 0xFF
            val b = pixel and 0xFF

            android.graphics.Color.RGBToHSV(r, g, b, hsv)
            hsv[1] = (hsv[1] * factor).coerceIn(0f, 1f) // Enhance saturation

            val enhancedColor = android.graphics.Color.HSVToColor(
                (pixel shr 24) and 0xFF, hsv
            )
            pixels[i] = enhancedColor
        }

        bitmap.setPixels(pixels, 0, width, 0, 0, width, height)
        return bitmap
    }

    /**
     * Determines whether light or dark text is more readable on a given background color.
     */
    @ColorInt
    private fun getBestTextColorForBackground(@ColorInt backgroundColor: Int, @ColorInt lightC
        val contrastWithLight = ColorUtils.calculateContrast(lightColor, backgroundColor)
        val contrastWithDark = ColorUtils.calculateContrast(darkColor, backgroundColor)
        return if (contrastWithLight > contrastWithDark) lightColor else darkColor
    }
}

// File: java\com\example\holodex\util\PlaylistFormatter.kt
// File: java/com/example/holodex/util/PlaylistFormatter.kt

package com.example.holodex.util

import android.content.Context
import com.example.holodex.R
import com.example.holodex.data.model.discovery.PlaylistStub
import com.google.gson.Gson
import com.google.gson.JsonSyntaxException
import timber.log.Timber

// Data classes to safely parse the JSON that is often in the description field of SGPs
private data class DescriptionContext(val channel: ChannelInfo?, val org: String?, val id: Str
private data class ChannelInfo(val name: String?, val english_name: String?)

// Base interface for all our specific formatters
private interface SgpFormatter {
    fun getTitle(
        playlist: PlaylistStub,
```

```
        params: Map<String, String>,
        descriptionJson: String?,
        context: Context,
        namePicker: (en: String?, jp: String?) -> String?
    ): String

    fun getDescription(
        playlist: PlaylistStub,
        params: Map<String, String>,
        descriptionJson: String?,
        context: Context,
        namePicker: (en: String?, jp: String?) -> String?
    ): String?
}

/**
 * A utility object to format playlist titles and descriptions, replicating
 * the logic from the Musicdex web frontend for consistency.
 */
object PlaylistFormatter {

    private val GSON = Gson()

    // The central map, mirroring the web app's `formatters` object
    private val formatters = mapOf<String, SgpFormatter>(
        ":artist" to ArtistFormatter,
        ":dailyrandom" to DailyRandomFormatter,
        ":video" to VideoFormatter,
        ":latest" to LatestFormatter,
        ":mv" to MvFormatter,
        ":weekly" to WeeklyFormatter,
        ":userweekly" to UserWeeklyFormatter,
        ":history" to HistoryFormatter,
        ":hot" to HotFormatter
    )

    /**
     * Gets the user-facing display title for any playlist.
     *
     * @param playlist The playlist object.
     * @param context Android context for string resources.
     * @param namePicker A helper lambda to choose between English and Japanese names.
     * @return The formatted title string.
     */
    fun getDisplayTitle(
        playlist: PlaylistStub,
        context: Context,
        namePicker: (en: String?, jp: String?) -> String?
    ): String {
        if (!playlist.id.startsWith(":")) {
            return playlist.title
        }

        val (type, params) = parsePlaylistID(playlist.id)
        val formatter = formatters[type] ?: DefaultFormatter
```

```
    return formatter.getTitle(playlist, params, playlist.description, context, namePicker)
}
/**
 * Gets the user-facing display description for any playlist.
 *
 * @param playlist The playlist object.
 * @param context Android context for string resources.
 * @param namePicker A helper lambda to choose between English and Japanese names.
 * @return The formatted description string, or null if there is none.
 */
fun getDisplayDescription(
    playlist: PlaylistStub,
    context: Context,
    namePicker: (en: String?, jp: String?) -> String?
): String? {
    if (!playlist.id.startsWith(":")) {
        return playlist.description
    }

    val (type, params) = parsePlaylistID(playlist.id)
    val formatter = formatters[type] ?: DefaultFormatter

    return formatter.getDescription(playlist, params, playlist.description, context, nameP
}
private fun parsePlaylistID(id: String): Pair<String, Map<String, String>> {
    if (!id.startsWith(":")) return Pair(id, emptyMap())
    val typeEndIndex = id.indexOf('[')
    if (typeEndIndex == -1) return Pair(id, emptyMap())

    val type = id.substring(0, typeEndIndex)
    val paramsString = id.substring(typeEndIndex + 1, id.lastIndexOf(']'))

    val params = paramsString.split(',')
        .mapNotNull {
            val parts = it.split('=', limit = 2)
            if (parts.size == 2) parts[0] to parts[1] else null
        }
        .toMap()

    return Pair(type, params)
}

private fun parseDescription(json: String?): DescriptionContext? {
    if (json.isNullOrBlank()) return null
    return try {
        GSON.fromJson(json, DescriptionContext::class.java)
    } catch (e: JsonSyntaxException) {
        Timber.e(e, "Failed to parse SGP description JSON: $json")
        null
    }
}

// --- Formatter Implementations ---
```

```
private object DefaultFormatter : SgpFormatter {
    override fun getTitle(p: PlaylistStub, pa: Map<String, String>, d: String?, c: Context
    override fun getDescription(p: PlaylistStub, pa: Map<String, String>, d: String?, c: C
}


private object ArtistFormatter : SgpFormatter {
    override fun getTitle(p: PlaylistStub, pa: Map<String, String>, d: String?, c: Context
        val channelInfo = parseDescription(d)?.channel
        val name = n(channelInfo?.english_name, channelInfo?.name) ?: "Artist"
        return c.getString(R.string.sgp_artist_radio_title, name)
    }
    override fun getDescription(p: PlaylistStub, pa: Map<String, String>, d: String?, c: C
        val channelInfo = parseDescription(d)?.channel
        val name = n(channelInfo?.english_name, channelInfo?.name) ?: "Artist"
        return c.getString(R.string.sgp_artist_radio_desc, name)
    }
}

private object DailyRandomFormatter : SgpFormatter {
    override fun getTitle(p: PlaylistStub, pa: Map<String, String>, d: String?, c: Context
        val channelInfo = parseDescription(d)?.channel
        val name = n(channelInfo?.english_name, channelInfo?.name) ?: "Artist"
        return c.getString(R.string.sgp_daily_mix_title, name)
    }
    override fun getDescription(p: PlaylistStub, pa: Map<String, String>, d: String?, c: C
        val channelInfo = parseDescription(d)?.channel
        val name = n(channelInfo?.english_name, channelInfo?.name) ?: "Artist"
        return c.getString(R.string.sgp_daily_mix_desc, name)
    }
}

private object MvFormatter : SgpFormatter {
    override fun getTitle(p: PlaylistStub, pa: Map<String, String>, d: String?, c: Context
        val org = pa["org"] ?: "Community"
        return when (pa["sort"]) {
            "random" -> c.getString(R.string.sgp_mv_random_title, org)
            "latest" -> c.getString(R.string.sgp_mv_latest_title, org)
            else -> p.title
        }
    }
    override fun getDescription(p: PlaylistStub, pa: Map<String, String>, d: String?, c: C
        val org = pa["org"] ?: "Community"
        return when (pa["sort"]) {
            "random" -> c.getString(R.string.sgp_mv_random_desc, org)
            "latest" -> c.getString(R.string.sgp_mv_latest_desc, org)
            else -> p.description
        }
    }
}

private object LatestFormatter : SgpFormatter {
    override fun getTitle(p: PlaylistStub, pa: Map<String, String>, d: String?, c: Context
        val org = pa["org"] ?: "Community"
        return c.getString(R.string.sgp_latest_title, org)
```

```
        }
        override fun getDescription(p: PlaylistStub, pa: Map<String, String>, d: String?, c: C
            val org = pa["org"] ?: "Community"
            return c.getString(R.string.sgp_latest_desc, org)
        }
    }

    private object WeeklyFormatter : SgpFormatter {
        override fun getTitle(p: PlaylistStub, pa: Map<String, String>, d: String?, c: Context
            val org = pa["org"] ?: "Community"
            return c.getString(R.string.sgp_weekly_mix_title, org)
        }
        override fun getDescription(p: PlaylistStub, pa: Map<String, String>, d: String?, c: C
            val org = pa["org"] ?: "Community"
            return c.getString(R.string.sgp_weekly_mix_desc, org)
        }
    }

    private object UserWeeklyFormatter : SgpFormatter {
        override fun getTitle(p: PlaylistStub, pa: Map<String, String>, d: String?, c: Context
        override fun getDescription(p: PlaylistStub, pa: Map<String, String>, d: String?, c: C
    }

    private object HistoryFormatter : SgpFormatter {
        override fun getTitle(p: PlaylistStub, pa: Map<String, String>, d: String?, c: Context
        override fun getDescription(p: PlaylistStub, pa: Map<String, String>, d: String?, c: C
    }

    private object HotFormatter : SgpFormatter {
        override fun getTitle(p: PlaylistStub, pa: Map<String, String>, d: String?, c: Context
        override fun getDescription(p: PlaylistStub, pa: Map<String, String>, d: String?, c: C
    }

    private object VideoFormatter: SgpFormatter {
        override fun getTitle(p: PlaylistStub, pa: Map<String, String>, d: String?, c: Context
            return parseDescription(d)?.title ?: p.title
        }
        override fun getDescription(p: PlaylistStub, pa: Map<String, String>, d: String?, c: C
            val desc = parseDescription(d)
            val name = n(desc?.channel?.english_name, desc?.channel?.name) ?: "Artist"
            return c.getString(R.string.sgp_video_desc, name)
        }
    }
}

// File: java\com\example\holodex\util\VideoFilteringUtil.kt
// File: java/com/example/holodex/util/VideoFilteringUtil.kt
package com.example.holodex.util

import com.example.holodex.data.model.HolodexVideoItem
import timber.log.Timber
import java.text.Normalizer

object VideoFilteringUtil {
```

```
private val STRICT_CORE_MUSIC_TOPICS = setOf("singing", "Music_Cover", "Original_Song")

private val musicKeywords = setOf(
    "cover", "?????", "song", "singing", "karaoke", "mv", "original song", "original", "mu
    "op/ed", "theme", "?", "??", "??????", "????????", "acoustic", "live", "concert",
    "????", "??", "medley", "arrange", "remix", "instrumental", "bgm", "soundtrack", "ost"
    "vocaloid", "???", "album", "single", "???", "guitar", "piano", "???", "??",
    "?", "???", "??????", "official audio", "music video"
)

private val channelMusicKeywords = setOf(
    "music", "song", "cover", "vsinger", "singer", "utaite", "archive", "records",
    "official channel", "??????", "??"
)

/**
 * Normalizes a title by converting all Unicode variants to their ASCII equivalents.
 * This includes:
 * - Full-width characters (Japanese/Chinese input style)
 * - Mathematical Alphanumeric Symbols (bold, italic, script, etc.)
 * - Enclosed Alphanumerics
 * - Accented characters
 * - Various stylized Unicode text
 *
 * Works for ANY word, not just hardcoded ones.
 */
private fun normalizeTitle(title: String): String {
    var normalized = title

    // Step 1: Replace common full-width punctuation
    normalized = normalized
        .replace("?", "(").replace("?", ")")
        .replace("?", "[").replace("?", "]")
        .replace("?", "{").replace("?", "}")
        .replace("?", "/").replace("?", "|")
        .replace("?", "@").replace("?", "#")
        .replace("?", "$").replace("?", "%")
        .replace("?", "&").replace("?", "*")
        .replace("?", "+").replace("?", "-")
        .replace("?", "=").replace("?", ":")
        .replace("?", ";").replace("?", "!")
        .replace("?", "?").replace("?", "~")
        .replace("?", "<").replace("?", ">")
        .replace("?", ".").replace("?", ",")
        .replace("'", "'").replace("'", "'")
        .replace("""", "\"").replace("""", "\"")
        .replace("?", " ") // Full-width space to regular space

    // Step 2: Use Unicode normalization to decompose accented characters
    // NFD = Canonical Decomposition (é becomes e + ´)
    normalized = Normalizer.normalize(normalized, Normalizer.Form.NFD)
        .replace(Regex("\\p{Mn}"), "") // Remove all diacritical marks

    // Step 3: Convert all stylized Unicode characters to ASCII
    val result = StringBuilder()
```

```
    var i = 0
    while (i < normalized.length) {
        val codePoint = normalized.codePointAt(i)
        val converted = convertUnicodeToAscii(codePoint)
        result.append(converted)
        i += Character.charCount(codePoint)
    }

    return result.toString()
}

/**
 * Converts a single Unicode code point to its ASCII equivalent string.
 * Handles multiple Unicode ranges for stylized text.
 */
private fun convertUnicodeToAscii(codePoint: Int): String {
    return when {
        // Full-width alphanumeric (?-?, ?-?, ?-?)
        codePoint in 0xFF01..0xFF5E -> {
            (codePoint - 0xFEE0).toChar().toString()
        }

        // Mathematical Bold (?-?, ?-?) U+1D400-U+1D433
        codePoint in 0x1D400..0x1D419 -> ('A'.code + (codePoint - 0x1D400)).toChar().toStr
        codePoint in 0x1D41A..0x1D433 -> ('a'.code + (codePoint - 0x1D41A)).toChar().toStr

        // Mathematical Italic (?-?, ?-?) U+1D434-U+1D467
        codePoint in 0x1D434..0x1D44D -> ('A'.code + (codePoint - 0x1D434)).toChar().toStr
        codePoint in 0x1D44E..0x1D467 -> ('a'.code + (codePoint - 0x1D44E)).toChar().toStr

        // Mathematical Bold Italic (?-?, ?-?) U+1D468-U+1D49B
        codePoint in 0x1D468..0x1D481 -> ('A'.code + (codePoint - 0x1D468)).toChar().toStr
        codePoint in 0x1D482..0x1D49B -> ('a'.code + (codePoint - 0x1D482)).toChar().toStr

        // Mathematical Script (?-?, ?-?) U+1D49C-U+1D4CF
        codePoint in 0x1D49C..0x1D4B5 -> ('A'.code + (codePoint - 0x1D49C)).toChar().toStr
        codePoint in 0x1D4B6..0x1D4CF -> ('a'.code + (codePoint - 0x1D4B6)).toChar().toStr

        // Mathematical Bold Script (?-?, ?-?) U+1D4D0-U+1D503
        codePoint in 0x1D4D0..0x1D4E9 -> ('A'.code + (codePoint - 0x1D4D0)).toChar().toStr
        codePoint in 0x1D4EA..0x1D503 -> ('a'.code + (codePoint - 0x1D4EA)).toChar().toStr

        // Mathematical Fraktur (?-?, ?-?) U+1D504-U+1D537
        codePoint in 0x1D504..0x1D51C -> ('A'.code + (codePoint - 0x1D504)).toChar().toStr
        codePoint in 0x1D51E..0x1D537 -> ('a'.code + (codePoint - 0x1D51E)).toChar().toStr

        // Mathematical Double-Struck (?-?, ?-?) U+1D538-U+1D56B
        codePoint in 0x1D538..0x1D550 -> ('A'.code + (codePoint - 0x1D538)).toChar().toStr
        codePoint in 0x1D552..0x1D56B -> ('a'.code + (codePoint - 0x1D552)).toChar().toStr

        // Mathematical Bold Fraktur (?-?, ?-?) U+1D56C-U+1D59F
        codePoint in 0x1D56C..0x1D585 -> ('A'.code + (codePoint - 0x1D56C)).toChar().toStr
        codePoint in 0x1D586..0x1D59F -> ('a'.code + (codePoint - 0x1D586)).toChar().toStr

        // Mathematical Sans-Serif (?-?, ?-?) U+1D5A0-U+1D5D3
```

```
            codePoint in 0x1D5A0..0x1D5B9 -> ('A'.code + (codePoint - 0x1D5A0)).toChar().toStr
            codePoint in 0x1D5BA..0x1D5D3 -> ('a'.code + (codePoint - 0x1D5BA)).toChar().toStr

            // Mathematical Sans-Serif Bold (?-?, ?-?) U+1D5D4-U+1D607
            codePoint in 0x1D5D4..0x1D5ED -> ('A'.code + (codePoint - 0x1D5D4)).toChar().toStr
            codePoint in 0x1D5EE..0x1D607 -> ('a'.code + (codePoint - 0x1D5EE)).toChar().toStr

            // Mathematical Sans-Serif Italic (?-?, ?-?) U+1D608-U+1D63B
            codePoint in 0x1D608..0x1D621 -> ('A'.code + (codePoint - 0x1D608)).toChar().toStr
            codePoint in 0x1D622..0x1D63B -> ('a'.code + (codePoint - 0x1D622)).toChar().toStr

            // Mathematical Sans-Serif Bold Italic (?-?, ?-?) U+1D63C-U+1D66F
            codePoint in 0x1D63C..0x1D655 -> ('A'.code + (codePoint - 0x1D63C)).toChar().toStr
            codePoint in 0x1D656..0x1D66F -> ('a'.code + (codePoint - 0x1D656)).toChar().toStr

            // Mathematical Monospace (?-?, ?-?) U+1D670-U+1D6A3
            codePoint in 0x1D670..0x1D689 -> ('A'.code + (codePoint - 0x1D670)).toChar().toStr
            codePoint in 0x1D68A..0x1D6A3 -> ('a'.code + (codePoint - 0x1D68A)).toChar().toStr

            // Enclosed Alphanumerics (?-?, ?-?) U+24B6-U+24E9
            codePoint in 0x24B6..0x24CF -> ('A'.code + (codePoint - 0x24B6)).toChar().toString
            codePoint in 0x24D0..0x24E9 -> ('a'.code + (codePoint - 0x24D0)).toChar().toString

            // Parenthesized Latin (?-?) U+249C-U+24B5
            codePoint in 0x249C..0x24B5 -> ('a'.code + (codePoint - 0x249C)).toChar().toString

            // Squared Latin (?-?, ?-?) U+1F130-U+1F149, U+1F170-U+1F189
            codePoint in 0x1F130..0x1F149 -> ('A'.code + (codePoint - 0x1F130)).toChar().toStr
            codePoint in 0x1F170..0x1F189 -> ('A'.code + (codePoint - 0x1F170)).toChar().toStr

            // Regional Indicator Symbols (?-?) U+1F1E6-U+1F1FF
            codePoint in 0x1F1E6..0x1F1FF -> ('A'.code + (codePoint - 0x1F1E6)).toChar().toStr

            else -> Character.toChars(codePoint).concatToString()
        }
    }

    fun isMusicContent(video: HolodexVideoItem): Boolean {
        val videoLogId = "${video.id} ('${video.title.take(30)}...')"
        Timber.d("isMusicContent Checking: ID=${videoLogId}, Type=${video.type}, Topic=${video

        // 1. Strongest Indicator: Positive Song Count
        if ((video.songcount ?: 0) > 0 || !video.songs.isNullOrEmpty()) {
            Timber.d("isMusicContent [PASS] ID=${videoLogId} via songcount > 0 or non-empty so
            return true
        }

        // 2. Strict Core Music Topics
        if (STRICT_CORE_MUSIC_TOPICS.contains(video.topicId)) {
            Timber.d("isMusicContent [PASS] ID=${videoLogId} via STRICT_CORE_MUSIC_TOPICS: ${v
            return true
        }

        // 3. Normalize and check title keywords
        val normalizedTitle = normalizeTitle(video.title).lowercase()
```

```
        if (musicKeywords.any { keyword -> normalizedTitle.contains(keyword) }) {
            Timber.d("isMusicContent [PASS] ID=${videoLogId} via musicKeyword in title (topic
            return true
        }

        // 4. Specific Check for Music Shorts
        if (video.topicId == "shorts" || (video.type == "clip" && video.duration > 0 && video.
            if (musicKeywords.any { keyword -> normalizedTitle.contains(keyword) }) {
                Timber.d("isMusicContent [PASS] ID=${videoLogId} via music short (type/duratio
                return true
            }
        }

        // 5. Fallback for Generic Topics
        val potentiallyMusicRelatedTopics = setOf("3D_Stream", "FreeChat", "??", "misc", "unkn
        if (potentiallyMusicRelatedTopics.contains(video.topicId)) {
            val channelNameLower = video.channel.name.lowercase()
            if (channelMusicKeywords.any { keyword -> channelNameLower.contains(keyword) }) {
                if (musicKeywords.any { keyword -> normalizedTitle.contains(keyword) }) {
                    Timber.d("isMusicContent [PASS] ID=${videoLogId} via generic topic, channe
                    return true
                }
            }
        }

        Timber.d("isMusicContent [FAIL] ID=${videoLogId}. No conditions met.")
        return false
    }
}

// File: java\com\example\holodex\viewmodel\ChannelDetailsViewModel.kt
// File: java/com/example/holodex/viewmodel/ChannelDetailsViewModel.kt
package com.example.holodex.viewmodel

import androidx.compose.ui.graphics.Color
import androidx.lifecycle.SavedStateHandle
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.example.holodex.data.db.ExternalChannelEntity
import com.example.holodex.data.model.discovery.ChannelDetails
import com.example.holodex.data.model.discovery.DiscoveryResponse
import com.example.holodex.data.repository.HolodexRepository
import com.example.holodex.data.repository.LocalRepository
import com.example.holodex.util.DynamicTheme
import com.example.holodex.util.PaletteExtractor
import com.example.holodex.viewmodel.mappers.toUnifiedDisplayItem
import com.example.holodex.viewmodel.mappers.toVideoShell
import com.example.holodex.viewmodel.state.UiState
import dagger.hilt.android.lifecycle.HiltViewModel
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.launch
import javax.inject.Inject
```

```
@HiltViewModel
class ChannelDetailsViewModel @Inject constructor(
    private val savedStateHandle: SavedStateHandle,
    private val holodexRepository: HolodexRepository,
    private val localRepository: LocalRepository,
    private val paletteExtractor: PaletteExtractor
) : ViewModel() {

    companion object {
        const val CHANNEL_ID_ARG = "channelId"
    }

    val channelId: String = savedStateHandle.get<String>(CHANNEL_ID_ARG) ?: ""

    private val _isExternal = MutableStateFlow(false)
    val isExternal: StateFlow<Boolean> = _isExternal.asStateFlow()

    private val _externalMusicItems = MutableStateFlow<UiState<List<UnifiedDisplayItem>>>(UiSt
    val externalMusicItems: StateFlow<UiState<List<UnifiedDisplayItem>>> = _externalMusicItems

    private val _channelDetailsState = MutableStateFlow<UiState<ChannelDetails>>(UiState.Loadi
    val channelDetailsState: StateFlow<UiState<ChannelDetails>> = _channelDetailsState.asState

    private val _dynamicTheme = MutableStateFlow(DynamicTheme.default(Color.Black, Color.White
    val dynamicTheme: StateFlow<DynamicTheme> = _dynamicTheme.asStateFlow()

    private val _discoveryState = MutableStateFlow<UiState<DiscoveryResponse>>(UiState.Loading
    val discoveryState: StateFlow<UiState<DiscoveryResponse>> = _discoveryState.asStateFlow()

    private val _popularSongsState = MutableStateFlow<UiState<List<UnifiedDisplayItem>>>(UiSta
    val popularSongsState: StateFlow<UiState<List<UnifiedDisplayItem>>> = _popularSongsState.a

    init {
        viewModelScope.launch {
            // Look up the channel in our local repository first
            val externalChannel = localRepository.getExternalChannel(channelId)

            if (externalChannel != null) {
                // It's an external channel
                _isExternal.value = true
                fetchChannelDetailsFromExternal(externalChannel)
                loadInitialPageFromExternalSource()
            } else {
                // It's a Holodex channel
                _isExternal.value = false
                loadAllHolodexContent()
            }
        }
    }

    private fun loadAllHolodexContent() {
        viewModelScope.launch {
            launch { fetchChannelDetails() }
            launch { fetchChannelDiscovery() }
```

```
        launch { fetchPopularSongs() }
    }
}

private fun fetchChannelDetailsFromExternal(channel: ExternalChannelEntity) {
    val details = ChannelDetails(
        id = channel.channelId,
        name = channel.name,
        englishName = channel.name,
        description = "Music from this channel is sourced directly from YouTube.",
        photoUrl = channel.photoUrl,
        bannerUrl = null, org = "External", suborg = null, twitter = null, group = null
    )
    _channelDetailsState.value = UiState.Success(details)
    viewModelScope.launch {
        _dynamicTheme.value = paletteExtractor.extractThemeFromUrl(
            channel.photoUrl,
            DynamicTheme.default(Color.Black, Color.White)
        )
    }
}

private fun loadInitialPageFromExternalSource() {
    viewModelScope.launch {
        _externalMusicItems.value = UiState.Loading
        holodexRepository.getMusicFromExternalChannel(channelId, null)
            .onSuccess { result ->
                val unifiedItems = result.data.map { it.toUnifiedDisplayItem(isLiked = fal
                _externalMusicItems.value = UiState.Success(unifiedItems)
            }.onFailure { _externalMusicItems.value = UiState.Error(it.localizedMessage ?:
    }
}

private suspend fun fetchChannelDetails() {
    holodexRepository.getChannelDetails(channelId)
        .onSuccess {
            _channelDetailsState.value = UiState.Success(it)
            _dynamicTheme.value = paletteExtractor.extractThemeFromUrl(
                it.bannerUrl,
                DynamicTheme.default(Color.Black, Color.White)
            )
        }
        .onFailure { _channelDetailsState.value = UiState.Error(it.localizedMessage ?: "Fa
}

private suspend fun fetchChannelDiscovery() {
    holodexRepository.getDiscoveryForChannel(channelId)
        .onSuccess { _discoveryState.value = UiState.Success(it) }
        .onFailure { _discoveryState.value = UiState.Error(it.localizedMessage ?: "Failed
}

private suspend fun fetchPopularSongs() {
    holodexRepository.getHotSongsForCarousel(channelId = channelId)
        .onSuccess { songs ->
            val displayItems = songs.map { song ->
```

```
                    val videoShell = song.toVideoShell()
                    song.toUnifiedDisplayItem(
                        parentVideo = videoShell,
                        isLiked = false,
                        isDownloaded = false
                    )
                }
                _popularSongsState.value = UiState.Success(displayItems)
            }
            .onFailure { _popularSongsState.value = UiState.Error(it.localizedMessage ?: "Fail
    }
}

// File: java\com\example\holodex\viewmodel\DiscoveryViewModel.kt
// File: java/com/example/holodex/viewmodel/DiscoveryViewModel.kt

package com.example.holodex.viewmodel

import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.example.holodex.auth.AuthState
import com.example.holodex.data.model.discovery.DiscoveryResponse
import com.example.holodex.data.model.discovery.PlaylistStub
import com.example.holodex.data.repository.HolodexRepository
import com.example.holodex.playback.PlaybackRequestManager
import com.example.holodex.playback.domain.repository.PlaybackRepository
import com.example.holodex.playback.domain.usecase.AddItemsToQueueUseCase
import com.example.holodex.viewmodel.mappers.toPlaybackItem
import com.example.holodex.viewmodel.mappers.toUnifiedDisplayItem
import com.example.holodex.viewmodel.mappers.toVideoShell
import com.example.holodex.viewmodel.state.UiState
import dagger.hilt.android.lifecycle.HiltViewModel
import kotlinx.coroutines.flow.MutableSharedFlow
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.SharedFlow
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asSharedFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.flow.update
import kotlinx.coroutines.launch
import timber.log.Timber
import javax.inject.Inject

enum class ShelfType {
    RECENT_STREAMS,
    SYSTEM_PLAYLISTS,
    ARTIST_RADIOS,
    FAN_PLAYLISTS,
    TRENDING_SONGS,
    DISCOVER_CHANNELS,
    FOR_YOU
}

data class DiscoveryScreenState(
    val shelves: Map<ShelfType, UiState<List<Any>>> = emptyMap(),
```

```
    val shelfOrder: List<ShelfType> = emptyList()
)

@HiltViewModel
class DiscoveryViewModel @Inject constructor(
    private val holodexRepository: HolodexRepository,
    private val playbackRequestManager: PlaybackRequestManager,
    private val playbackRepository: PlaybackRepository,
    private val addItemsToQueueUseCase: AddItemsToQueueUseCase
) : ViewModel() {

    private val _uiState = MutableStateFlow(DiscoveryScreenState())
    val uiState: StateFlow<DiscoveryScreenState> = _uiState.asStateFlow()

    private val _forYouState = MutableStateFlow<UiState<DiscoveryResponse>>(UiState.Loading)
    val forYouState: StateFlow<UiState<DiscoveryResponse>> = _forYouState.asStateFlow()

    private val _transientMessage = MutableSharedFlow<String>()
    val transientMessage: SharedFlow<String> = _transientMessage.asSharedFlow()

    fun loadDiscoveryContent(organization: String, authState: AuthState) {
        val isFavoritesView = organization == "Favorites"
        val newShelfOrder = if (isFavoritesView && authState is AuthState.LoggedIn) {
            listOf(ShelfType.FOR_YOU)
        } else {
            listOf(
                ShelfType.RECENT_STREAMS,
                ShelfType.SYSTEM_PLAYLISTS,
                ShelfType.ARTIST_RADIOS,
                ShelfType.FAN_PLAYLISTS,
                ShelfType.TRENDING_SONGS,
                ShelfType.DISCOVER_CHANNELS
            )
        }

        val initialShelves = newShelfOrder.associateWith { UiState.Loading }
        _uiState.value = DiscoveryScreenState(shelves = initialShelves, shelfOrder = newShelfO

        if (isFavoritesView && authState is AuthState.LoggedIn) {
            fetchFavoritesHub()
        } else {
            val orgParam = organization.takeIf { it != "All Vtubers" }
            fetchTrendingSongs(orgParam)
            fetchDiscoveryHub(organization)
        }
    }

    fun loadForYouContent() {
        _forYouState.value = UiState.Loading
        viewModelScope.launch {
            holodexRepository.getFavoritesHubContent()
                .onSuccess { response ->
                    _forYouState.value = UiState.Success(response)
                }
                .onFailure { error ->
```

```
                _forYouState.value = UiState.Error(error.localizedMessage ?: "Failed to lo
            }
        }
    }

    private fun fetchTrendingSongs(organization: String?) {
        viewModelScope.launch {
            holodexRepository.getHotSongsForCarousel(org = organization)
                .onSuccess { songs ->
                    val displayItems = songs.map { song ->
                        val videoShell = song.toVideoShell()
                        song.toUnifiedDisplayItem(
                            parentVideo = videoShell,
                            isLiked = false,
                            isDownloaded = false
                        )
                    }
                    _uiState.update { s -> s.copy(shelves = s.shelves + (ShelfType.TRENDING_SO
                }.onFailure { e ->
                    _uiState.update { s -> s.copy(shelves = s.shelves + (ShelfType.TRENDING_SO
                }
        }
    }

    private fun fetchDiscoveryHub(org: String) {
        viewModelScope.launch {
            holodexRepository.getDiscoveryHubContent(org)
                .onSuccess { response ->
                    val allPlaylists = response.recommended?.playlists ?: emptyList()

                    val systemPlaylists = allPlaylists.filter { it.type.startsWith("playlist/"
                    val radios = allPlaylists.filter { it.type.startsWith("radio/") }
                    val communityPlaylists = allPlaylists.filter { it.type == "ugp" }
                    val recentStreams = response.recentSingingStreams?.filter {
                        it.playlist?.content?.isNotEmpty() == true
                    } ?: emptyList()
                    val discoverChannels = response.channels ?: emptyList()

                    _uiState.update { s ->
                        s.copy(
                            shelves = s.shelves +
                                    (ShelfType.RECENT_STREAMS to UiState.Success(recentStreams
                                    (ShelfType.SYSTEM_PLAYLISTS to UiState.Success(systemPlayl
                                    (ShelfType.ARTIST_RADIOS to UiState.Success(radios)) +
                                    (ShelfType.FAN_PLAYLISTS to UiState.Success(communityPlayl
                                    (ShelfType.DISCOVER_CHANNELS to UiState.Success(discoverCh
                        )
                    }
                }.onFailure { e ->
                    val errorState = UiState.Error(e.localizedMessage ?: "Error")
                    _uiState.update { s ->
                        s.copy(
                            shelves = s.shelves +
                                    (ShelfType.RECENT_STREAMS to errorState) +
                                    (ShelfType.SYSTEM_PLAYLISTS to errorState) +
```

```
                                (ShelfType.ARTIST_RADIOS to errorState) +
                                (ShelfType.FAN_PLAYLISTS to errorState) +
                                (ShelfType.DISCOVER_CHANNELS to errorState)
                        )
                    }
                }
            }
        }

        private fun fetchFavoritesHub() {
            viewModelScope.launch {
                holodexRepository.getFavoritesHubContent()
                    .onSuccess { response ->
                        val recentStreams = response.recentSingingStreams?.filter {
                            it.playlist?.content?.isNotEmpty() == true
                        } ?: emptyList()
                        _uiState.update { s -> s.copy(shelves = s.shelves + (ShelfType.FOR_YOU to
                    }.onFailure { e ->
                        _uiState.update { s -> s.copy(shelves = s.shelves + (ShelfType.FOR_YOU to
                    }
            }
        }

        fun playUnifiedItem(item: UnifiedDisplayItem) {
            viewModelScope.launch {
                playbackRequestManager.submitPlaybackRequest(items = listOf(item.toPlaybackItem())
            }
        }
        fun playRadioPlaylist(playlist: PlaylistStub) {
            viewModelScope.launch {
                if (playlist.type.startsWith("radio")) {
                    Timber.d("Playing playlist as Radio: ${playlist.id}")
                    playbackRepository.prepareAndPlayRadio(playlist.id)
                } else {
                    // This is a fallback for playing a normal playlist from this screen
                    val result = holodexRepository.getFullPlaylistContent(playlist.id)
                    result.onSuccess { fullPlaylist ->
                        val playbackItems = fullPlaylist.content?.mapNotNull { song ->
                            if (song.channel.id == null) {
                                null
                            } else {
                                val videoShell = song.toVideoShell(fullPlaylist.title)
                                song.toPlaybackItem(videoShell)
                            }
                        } ?: emptyList()

                        if (playbackItems.isNotEmpty()) {
                            playbackRequestManager.submitPlaybackRequest(items = playbackItems)
                        } else {
                            _transientMessage.emit("This playlist appears to be empty.")
                        }
                    }.onFailure { error ->
                        _transientMessage.emit("Error: Could not load playlist.")
                    }
                }
```

```
        }
    }
    fun addAllToQueue(items: List<UnifiedDisplayItem>) {
        viewModelScope.launch {
            if (items.isNotEmpty()) {
                val playbackItems = items.map { it.toPlaybackItem() }
                addItemsToQueueUseCase(playbackItems)
                _transientMessage.emit("Added ${items.size} songs to queue.")
            }
        }
    }
}

// File: java\com\example\holodex\viewmodel\DownloadsViewModel.kt
// File: java/com/example/holodex/viewmodel/DownloadsViewModel.kt

package com.example.holodex.viewmodel

import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import androidx.media3.common.util.UnstableApi
import com.example.holodex.data.db.DownloadStatus
import com.example.holodex.data.db.DownloadedItemEntity
import com.example.holodex.data.db.LikedItemType
import com.example.holodex.data.repository.DownloadRepository
import com.example.holodex.data.repository.HolodexRepository
import com.example.holodex.playback.PlaybackRequestManager
import com.example.holodex.playback.domain.model.PlaybackItem
import dagger.hilt.android.lifecycle.HiltViewModel
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.SharingStarted
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.flow.combine
import kotlinx.coroutines.flow.stateIn
import kotlinx.coroutines.launch
import timber.log.Timber
import javax.inject.Inject

@UnstableApi
@HiltViewModel
class DownloadsViewModel @UnstableApi
@Inject constructor(
    private val downloadRepository: DownloadRepository,
    private val holodexRepository: HolodexRepository,
    private val playbackRequestManager: PlaybackRequestManager
) : ViewModel() {

    companion object {
        private const val TAG = "DownloadsViewModel"
    }

    private val allDownloads: StateFlow<List<DownloadedItemEntity>> =
        downloadRepository.getAllDownloads()
            .stateIn(
```

```
            scope = viewModelScope,
            started = SharingStarted.WhileSubscribed(5000),
            initialValue = emptyList()
        )

    private val _searchQuery = MutableStateFlow("")
    val searchQuery: StateFlow<String> = _searchQuery.asStateFlow()

    val filteredDownloads: StateFlow<List<DownloadedItemEntity>> =
        combine(allDownloads, _searchQuery) { downloads, query ->
            if (query.isBlank()) {
                downloads
            } else {
                downloads.filter {
                    it.title.contains(query, ignoreCase = true) ||
                            it.artistText.contains(query, ignoreCase = true)
                }
            }
        }.stateIn(
            scope = viewModelScope,
            started = SharingStarted.WhileSubscribed(5000),
            initialValue = emptyList()
        )

    fun onSearchQueryChanged(query: String) {
        _searchQuery.value = query
    }

    fun retryDownload(item: DownloadedItemEntity) {
        viewModelScope.launch {
            try {
                Timber.d("$TAG: Retrying download for item: ${item.videoId}")
                val videoId = item.videoId.substringBeforeLast('_')
                val songStart = item.videoId.substringAfterLast('_').toIntOrNull()
                Timber.d("$TAG: Retrying download for item: ${item.videoId} (Parent: $videoId,

                if (songStart == null) {
                    Timber.e("$TAG: Cannot retry, invalid item ID format: ${item.videoId}")
                    return@launch
                }

                val result = holodexRepository.getVideoWithSongs(videoId, forceRefresh = true)
                result.onSuccess { videoWithSongs ->
                    val songToRetry = videoWithSongs.songs?.find { it.start == songStart }
                    if (songToRetry != null) {
                        Timber.i("$TAG: Found matching song to retry: '${songToRetry.name}'. S
                        downloadRepository.startDownload(videoWithSongs, songToRetry)
                    } else {
                        Timber.e("$TAG: Could not find matching song with start time $songStar
                    }
                }.onFailure { exception ->
                    Timber.e(exception, "$TAG: Failed to fetch video details for retry.")
                }
            } catch (e: Exception) {
                Timber.e(e, "$TAG: Error during retry download for ${item.videoId}")
```

```
        }
    }
}


fun playDownloads(tappedItem: DownloadedItemEntity) {
    if (tappedItem.downloadStatus != DownloadStatus.COMPLETED) {
        Timber.w("$TAG: Attempted to play a non-completed download: ${tappedItem.videoId}.
        return
    }
    if (tappedItem.localFileUri.isNullOrBlank()) {
        Timber.e("$TAG: Tapped item ${tappedItem.videoId} is completed but has no local fi
        return
    }

    viewModelScope.launch {
        val currentVisibleDownloads = filteredDownloads.value
            .filter { it.downloadStatus == DownloadStatus.COMPLETED }

        val playableDownloads = currentVisibleDownloads.filter { !it.localFileUri.isNullOr

        if (playableDownloads.isEmpty()) {
            Timber.e("$TAG: Play request initiated, but the visible download list is empty
            return@launch
        }

        val playbackItems = playableDownloads.map { mapDownloadToPlaybackItem(it) }
        val startIndex = playbackItems.indexOfFirst { it.id == tappedItem.videoId }.coerce
        Timber.i("$TAG: Playing downloads queue. Tapped index: $startIndex, Total items in
        playbackRequestManager.submitPlaybackRequest(playbackItems, startIndex)
    }
}

fun playAllDownloadsShuffled() {
    viewModelScope.launch {
        val completedDownloads = filteredDownloads.value
            .filter { it.downloadStatus == DownloadStatus.COMPLETED }

        val playableDownloads = completedDownloads.filter { !it.localFileUri.isNullOrBlank

        if (playableDownloads.isNotEmpty()) {
            val playbackItems = playableDownloads.map { mapDownloadToPlaybackItem(it) }
            playbackRequestManager.submitPlaybackRequest(playbackItems, 0, shouldShuffle =
        } else {
            Timber.w("$TAG: No playable downloads found for shuffle playback.")
        }
    }
}

fun deleteDownload(itemId: String) {
    viewModelScope.launch {
        Timber.d("$TAG: Deleting download with ID: $itemId")
        try {
            downloadRepository.deleteDownloadById(itemId)
        } catch (e: Exception) {
```

```
            Timber.e(e, "Failed to delete download: $itemId")
        }
    }
}

@UnstableApi
fun cancelDownload(videoId: String) {
    viewModelScope.launch {
        try {
            Timber.d("$TAG: Cancelling download for videoId: $videoId")
            downloadRepository.cancelDownload(videoId)
        } catch (e: Exception) {
            Timber.e(e, "Failed to cancel download for $videoId")
        }
    }
}

fun resumeDownload(videoId: String) {
    viewModelScope.launch {
        try {
            Timber.d("$TAG: Resuming download for videoId: $videoId")
            downloadRepository.resumeDownload(videoId)
        } catch (e: Exception) {
            Timber.e(e, "Failed to resume download for $videoId")
        }
    }
}

fun purgeStaleDownloads() {
    viewModelScope.launch {
        try {
            Timber.d("$TAG: Purging stale downloads (now running full reconciliation)")
            downloadRepository.reconcileAllDownloads()
        } catch (e: Exception) {
            Timber.e(e, "Failed to reconcile downloads")
        }
    }
}

/**
 * Relays the request to re-trigger the AudioProcessingWorker to the repository.
 * This is for items that have been successfully downloaded but failed during post-process
 */
fun retryExport(item: DownloadedItemEntity) {
    viewModelScope.launch {
        Timber.d("$TAG: Relaying retry export request for ${item.videoId} to repository.")
        try {
            downloadRepository.retryExportForItem(item)
        } catch (e: Exception) {
            Timber.e(e, "Failed to relay retry export for ${item.videoId}")
        }
    }
}
fun mapDownloadToPlaybackItem(item: DownloadedItemEntity): PlaybackItem {
    val parentVideoId = item.videoId.substringBeforeLast('_')
```

```
        val songStartSec = item.videoId.substringAfterLast('_').toLongOrNull() ?: 0
        return PlaybackItem(
            id = item.videoId,
            videoId = parentVideoId,
            songId = item.videoId,
            serverUuid = item.videoId, // The ID for a download IS the server's unique ID for
            title = item.title,
            artistText = item.artistText,
            albumText = item.title,
            artworkUri = item.artworkUrl,
            durationSec = item.durationSec,
            streamUri = item.localFileUri,
            clipStartSec = songStartSec,
            clipEndSec = songStartSec + item.durationSec,
            description = null,
            channelId = item.channelId,
            originalArtist = item.artistText
        )
    }
}
fun PlaybackItem.toUnifiedDisplayItem(): UnifiedDisplayItem {
    return UnifiedDisplayItem(
        stableId = "download_${this.id}",
        playbackItemId = this.id,
        videoId = this.videoId,
        channelId = this.channelId,
        title = this.title,
        artistText = this.artistText,
        artworkUrls = listOfNotNull(this.artworkUri),
        durationText = com.example.holodex.playback.util.formatDurationSeconds(this.durationSe
        isSegment = true,
        songCount = null,
        isDownloaded = true,
        isLiked = false, // We don't have this info here, FavoritesViewModel will provide it
        itemTypeForPlaylist = LikedItemType.SONG_SEGMENT,
        songStartSec = this.clipStartSec?.toInt(),
        songEndSec = this.clipEndSec?.toInt(),
        originalArtist = this.originalArtist,
        isExternal = false // Downloads are never external
    )
}

// File: java\com\example\holodex\viewmodel\ExternalChannelViewModel.kt
// File: java/com/example/holodex/viewmodel/ExternalChannelViewModel.kt
package com.example.holodex.viewmodel

import androidx.compose.ui.graphics.Color
import androidx.lifecycle.SavedStateHandle
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.example.holodex.data.db.ExternalChannelEntity
import com.example.holodex.data.model.ChannelSearchResult
import com.example.holodex.data.model.discovery.ChannelDetails
import com.example.holodex.data.repository.HolodexRepository
import com.example.holodex.data.repository.LocalRepository
```

```
import com.example.holodex.util.DynamicTheme
import com.example.holodex.util.PaletteExtractor
import com.example.holodex.viewmodel.mappers.toUnifiedDisplayItem
import com.example.holodex.viewmodel.state.UiState
import dagger.hilt.android.lifecycle.HiltViewModel
import kotlinx.coroutines.FlowPreview
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.flow.debounce
import kotlinx.coroutines.launch
import org.schabi.newpipe.extractor.Page
import javax.inject.Inject

@OptIn(FlowPreview::class)
@HiltViewModel
class ExternalChannelViewModel @Inject constructor(
    private val savedStateHandle: SavedStateHandle,
    private val holodexRepository: HolodexRepository,
    private val localRepository: LocalRepository,
    private val paletteExtractor: PaletteExtractor
) : ViewModel() {

    val channelId: String = savedStateHandle.get<String>("channelId") ?: ""

    // States for the "Add Channel" dialog
    private val _showDialog = MutableStateFlow(false)
    val showDialog: StateFlow<Boolean> = _showDialog.asStateFlow()
    private val _searchQuery = MutableStateFlow("")
    val searchQuery: StateFlow<String> = _searchQuery.asStateFlow()
    private val _searchState = MutableStateFlow<UiState<List<ChannelSearchResult>>>(UiState.Su
    val searchState: StateFlow<UiState<List<ChannelSearchResult>>> = _searchState.asStateFlow(
    private val _isAdding = MutableStateFlow<Set<String>>(emptySet())
    val isAdding: StateFlow<Set<String>> = _isAdding.asStateFlow()

    // States for the "External Channel Details" screen
    private val _channelDetails = MutableStateFlow<ChannelDetails?>(null)
    val channelDetails: StateFlow<ChannelDetails?> = _channelDetails.asStateFlow()

    private val _musicItems = MutableStateFlow<List<UnifiedDisplayItem>>(emptyList())
    val musicItems: StateFlow<List<UnifiedDisplayItem>> = _musicItems.asStateFlow()

    private val _uiState = MutableStateFlow<UiState<Unit>>(UiState.Loading)
    val uiState: StateFlow<UiState<Unit>> = _uiState.asStateFlow()

    val dynamicTheme: StateFlow<DynamicTheme> get() = _dynamicTheme
    private val _dynamicTheme = MutableStateFlow(DynamicTheme.default(Color.Black, Color.White

    // --- PAGINATION STATE - NOW PUBLIC ---
    private val _isLoadingMore = MutableStateFlow(false)
    val isLoadingMore: StateFlow<Boolean> = _isLoadingMore.asStateFlow()

    private val _endOfList = MutableStateFlow(false)
    val endOfList: StateFlow<Boolean> = _endOfList.asStateFlow()
```

```
private var nextPageCursor: Page? = null

init {
    if (channelId.isNotBlank()) {
        loadInitialDataForScreen()
    }
    viewModelScope.launch {
        _searchQuery.debounce(500).collect { query ->
            if (query.length > 2) {
                performChannelSearch(query)
            } else if (query.isEmpty()) {
                _searchState.value = UiState.Success(emptyList())
            }
        }
    }
}

private fun loadInitialDataForScreen() {
    viewModelScope.launch {
        val channel = localRepository.getExternalChannel(channelId)
        if (channel != null) {
            _channelDetails.value = ChannelDetails(
                id = channel.channelId, name = channel.name, englishName = channel.name,
                description = "Music from this channel is sourced directly from YouTube.",
                photoUrl = channel.photoUrl, bannerUrl = null, org = "External",
                suborg = null, twitter = null, group = null
            )
            _dynamicTheme.value = paletteExtractor.extractThemeFromUrl(
                channel.photoUrl, DynamicTheme.default(Color.Black, Color.White)
            )
            loadMoreMusic(isInitialLoad = true)
        } else {
            _uiState.value = UiState.Error("Channel not found in local library.")
        }
    }
}

fun loadMoreMusic(isInitialLoad: Boolean = false) {
    if (_isLoadingMore.value || (_endOfList.value && !isInitialLoad)) return

    viewModelScope.launch {
        _isLoadingMore.value = true
        if (isInitialLoad) {
            _uiState.value = UiState.Loading
            _musicItems.value = emptyList()
            nextPageCursor = null
            _endOfList.value = false
        }

        holodexRepository.getMusicFromExternalChannel(channelId, nextPageCursor)
            .onSuccess { result ->
                val newItems = result.data.map { it.toUnifiedDisplayItem(isLiked = false,
                _musicItems.value += newItems
                nextPageCursor = result.nextPageCursor as? Page
                if (nextPageCursor == null) {
```

```
                        _endOfList.value = true
                    }
                    _uiState.value = UiState.Success(Unit)
                }.onFailure {
                    _uiState.value = UiState.Error(it.localizedMessage ?: "Failed to load musi
                }
            _isLoadingMore.value = false
        }
    }

    // --- Logic for "Add Channel" Dialog ---
    fun openDialog() { _showDialog.value = true }
    fun closeDialog() {
        _showDialog.value = false
        _searchQuery.value = ""
        _searchState.value = UiState.Success(emptyList())
    }
    fun onSearchQueryChanged(query: String) { _searchQuery.value = query }
    private fun performChannelSearch(query: String) {
        viewModelScope.launch {
            _searchState.value = UiState.Loading
            holodexRepository.searchForExternalChannels(query)
                .onSuccess { results -> _searchState.value = UiState.Success(results) }
                .onFailure { error -> _searchState.value = UiState.Error(error.localizedMessag
        }
    }
    fun addChannel(channel: ChannelSearchResult) {
        viewModelScope.launch {
            _isAdding.value += channel.channelId
            val entity = ExternalChannelEntity(
                channelId = channel.channelId, name = channel.name, photoUrl = channel.thumbna
            )
            localRepository.addExternalChannel(entity)
        }
    }
}

// File: java\com\example\holodex\viewmodel\FavoritesViewModel.kt
// File: java/com/example/holodex/viewmodel/FavoritesViewModel.kt
package com.example.holodex.viewmodel

import androidx.lifecycle.ViewModel
import androidx.media3.common.util.UnstableApi
import com.example.holodex.data.db.FavoriteChannelEntity
import com.example.holodex.data.db.LikedItemDao
import com.example.holodex.data.db.LikedItemEntity
import com.example.holodex.data.db.LikedItemType
import com.example.holodex.data.db.LocalFavoriteEntity
import com.example.holodex.data.db.SyncStatus
import com.example.holodex.data.model.HolodexChannelMin
import com.example.holodex.data.model.HolodexSong
import com.example.holodex.data.model.HolodexVideoItem
import com.example.holodex.data.model.discovery.ChannelDetails
import com.example.holodex.data.repository.DownloadRepository
import com.example.holodex.data.repository.HolodexRepository
```

```
import com.example.holodex.data.repository.LocalRepository
import com.example.holodex.playback.domain.model.PlaybackItem
import com.example.holodex.playback.util.formatDurationSeconds
import com.example.holodex.viewmodel.mappers.toUnifiedDisplayItem
import dagger.hilt.android.lifecycle.HiltViewModel
import kotlinx.coroutines.flow.combine
import kotlinx.coroutines.flow.map
import org.orbitmvi.orbit.Container
import org.orbitmvi.orbit.ContainerHost
import org.orbitmvi.orbit.viewmodel.container
import timber.log.Timber
import javax.inject.Inject

// --- State, SideEffect, and Helper Enums ---
enum class StorageLocation { LIKED_ITEMS, LOCAL_FAVORITES }
enum class ItemCategory { SYNCABLE_SEGMENT, VIRTUAL_SEGMENT, FULL_VIDEO }

data class FavoritesState(
    val likedItemsMap: Map<String, StorageLocation> = emptyMap(),
    val unifiedLikedSegments: List<UnifiedDisplayItem> = emptyList(),
    val unifiedFavoritedVideos: List<UnifiedDisplayItem> = emptyList(),
    val favoriteChannels: List<Any> = emptyList(),
    val isLoading: Boolean = true
)

sealed class FavoritesSideEffect {
    data class ShowToast(val message: String) : FavoritesSideEffect()
}

@UnstableApi
@HiltViewModel
class FavoritesViewModel @Inject constructor(
    private val holodexRepository: HolodexRepository,
    private val localRepository: LocalRepository,
    private val likedItemDao: LikedItemDao,
    private val downloadRepository: DownloadRepository
) : ViewModel(), ContainerHost<FavoritesState, FavoritesSideEffect> {

    override val container: Container<FavoritesState, FavoritesSideEffect> =
        container(FavoritesState()) {
            // This is the "onCreate" block. We start an intent that will run for the ViewMode
            intent {
                // repeatOnSubscription is the key for collecting flows in a lifecycle-aware w
                // It starts collecting when the UI is visible and stops when it's not.
                repeatOnSubscription {
                    val likedItemsFlow = combine(
                        holodexRepository.getObservableLikedSongSegments(),
                        holodexRepository.getFavoritedVideosPaged(0, 1000),
                        localRepository.getLocalFavorites()
                    ) { syncedSegments, syncedVideos, localFavs ->
                        Triple(syncedSegments, syncedVideos, localFavs)
                    }

                    val channelFlow = combine(
                        holodexRepository.getFavoriteChannels(),
```

```
                localRepository.getAllExternalChannels()
            ) { syncedChannels, localChannels ->
                syncedChannels to localChannels
            }

            combine(
                likedItemsFlow,
                channelFlow,
                downloadRepository.getAllDownloads()
                    .map { list -> list.map { it.videoId }.toSet() }
            ) { (syncedSegments, syncedVideos, localFavs), (syncedChannels, localChann

                val map = buildMap {
                    syncedSegments.forEach { put(it.itemId, StorageLocation.LIKED_ITEM
                    syncedVideos.forEach { put(it.itemId, StorageLocation.LIKED_ITEMS)
                    localFavs.forEach { put(it.itemId, StorageLocation.LOCAL_FAVORITES
                }

                val unifiedSegments = (
                        syncedSegments.map {
                            it.toUnifiedDisplayItem(
                                downloadedIds.contains(
                                    it.itemId
                                )
                            )
                        } +
                                localFavs.filter { it.isSegment }
                                    .map { it.toUnifiedDisplayItem(downloadedIds.conta
                        ).sortedByDescending { it.stableId }

                val unifiedVideos = (
                        syncedVideos.map { it.toUnifiedDisplayItem(false) } +
                                localFavs.filter { !it.isSegment }
                                    .map { it.toUnifiedDisplayItem(false) }
                        ).sortedByDescending { it.stableId }

                val unifiedChannels =
                    (syncedChannels + localChannels).sortedByDescending { if (it is Fa

                // This transformation creates the new state object
                FavoritesState(
                    likedItemsMap = map,
                    unifiedLikedSegments = unifiedSegments,
                    unifiedFavoritedVideos = unifiedVideos,
                    favoriteChannels = unifiedChannels,
                    isLoading = false // Data has arrived
                )
            }.collect { newState ->
                // This reduce block is now correctly inside the intent's scope.
                reduce { newState }
            }
        }
    }
}
```

```
fun toggleLike(item: PlaybackItem) = intent {
    try {
        val likeId = getLikeIdForPlaybackItem(item)
        val storageLocation = state.likedItemsMap[likeId]

        if (storageLocation != null) {
            // UNLIKE PATH
            when (storageLocation) {
                StorageLocation.LIKED_ITEMS -> holodexRepository.removeLikedItem(likeId)
                StorageLocation.LOCAL_FAVORITES -> localRepository.removeLocalFavorite(lik
            }
            postSideEffect(FavoritesSideEffect.ShowToast("Removed from favorites"))
        } else {
            // LIKE PATH
            when (categorizeItem(item)) {
                ItemCategory.SYNCABLE_SEGMENT -> addSyncedSongSegment(item)
                ItemCategory.VIRTUAL_SEGMENT -> addLocalVirtualSegment(item)
                ItemCategory.FULL_VIDEO -> addLocalHolodexVideo(item)
            }
            postSideEffect(FavoritesSideEffect.ShowToast("Added to favorites"))
        }
    } catch (e: Exception) {
        Timber.e(e, "Failed to toggle like for item: ${item.title}")
        postSideEffect(FavoritesSideEffect.ShowToast("Error: Could not update favorite"))
    }
}

fun toggleFavoriteChannel(videoItem: HolodexVideoItem) = intent {
    val channelId = videoItem.channel.id ?: return@intent
    val isFavorited =
        state.favoriteChannels.any { (it is FavoriteChannelEntity && it.id == channelId) }
    if (isFavorited) {
        holodexRepository.removeFavoriteChannel(channelId)
    } else {
        holodexRepository.addFavoriteChannel(videoItem)
    }
}

fun toggleFavoriteChannelByDetails(details: ChannelDetails) = intent {
    val isFavorited =
        state.favoriteChannels.any { (it is FavoriteChannelEntity && it.id == details.id)
    if (isFavorited) {
        holodexRepository.removeFavoriteChannel(details.id)
    } else {
        val videoShell = HolodexVideoItem(
            id = "channel_favorite_${details.id}", title = details.name, type = "placehold
            topicId = null, availableAt = "", publishedAt = null, duration = 0, status = "
            channel = HolodexChannelMin(
                id = details.id, name = details.name, englishName = details.englishName,
                org = details.org, type = "vtuber", photoUrl = details.photoUrl
            ), songcount = null, description = details.description, songs = null
        )
        holodexRepository.addFavoriteChannel(videoShell)
    }
}
```

```
fun getLikeIdForPlaybackItem(item: PlaybackItem): String {
    return if (item.clipStartSec != null) {
        LikedItemEntity.generateSongItemId(item.videoId, item.clipStartSec.toInt())
    } else {
        LikedItemEntity.generateVideoItemId(item.videoId)
    }
}

private fun categorizeItem(item: PlaybackItem): ItemCategory {
    return when {
        item.isExternal -> ItemCategory.VIRTUAL_SEGMENT
        item.clipStartSec != null -> {
            if (item.clipStartSec > 0) ItemCategory.SYNCABLE_SEGMENT
            else ItemCategory.VIRTUAL_SEGMENT
        }

        else -> ItemCategory.FULL_VIDEO
    }
}

private suspend fun addSyncedSongSegment(item: PlaybackItem) {
    try {
        val itemId = getLikeIdForPlaybackItem(item)
        val existingItem = likedItemDao.getLikedItem(itemId)
        if (existingItem?.syncStatus == SyncStatus.PENDING_DELETE) {
            likedItemDao.updateStatusAndTimestamp(
                itemId,
                SyncStatus.DIRTY,
                System.currentTimeMillis()
            )
        } else if (existingItem == null) {
            val songForLike = HolodexSong(
                name = item.title, start = item.clipStartSec!!.toInt(),
                end = item.clipEndSec?.toInt() ?: 0,
                itunesId = null, artUrl = item.artworkUri, videoId = item.videoId,
                originalArtist = item.originalArtist
            )
            val videoContext = HolodexVideoItem(
                id = item.videoId,
                title = item.albumText ?: item.title,
                type = "stream",
                topicId = null,
                publishedAt = null,
                availableAt = "",
                duration = item.durationSec,
                status = "past",
                channel = HolodexChannelMin(
                    id = item.channelId, name = item.artistText, englishName = null,
                    org = null, type = "vtuber", photoUrl = null
                ),
                songcount = null,
                description = item.description,
                songs = null
            )
```

```
            holodexRepository.addLikedSongSegment(videoContext, songForLike)
        }
    } catch (e: Exception) {
        Timber.e(e, "Failed to add synced song segment")
        intent { postSideEffect(FavoritesSideEffect.ShowToast("Error: ${e.message}")) }
    }
}

private suspend fun addLocalVirtualSegment(item: PlaybackItem) {
    try {
        val localFavorite = LocalFavoriteEntity(
            itemId = getLikeIdForPlaybackItem(item),
            videoId = item.videoId,
            channelId = item.channelId,
            title = item.title,
            artistText = item.artistText,
            artworkUrl = item.artworkUri,
            durationSec = item.durationSec,
            isSegment = true,
            songStartSec = item.clipStartSec?.toInt(),
            songEndSec = item.clipEndSec?.toInt()
        )
        localRepository.addLocalFavorite(localFavorite)
    } catch (e: Exception) {
        Timber.e(e, "Failed to add local virtual segment like")
    }
}

private suspend fun addLocalHolodexVideo(item: PlaybackItem) {
    try {
        val entity = LikedItemEntity(
            itemId = getLikeIdForPlaybackItem(item), videoId = item.videoId,
            itemType = LikedItemType.VIDEO, serverId = null, titleSnapshot = item.title,
            artistTextSnapshot = item.artistText, albumTextSnapshot = item.albumText,
            artworkUrlSnapshot = item.artworkUri, descriptionSnapshot = item.description,
            channelIdSnapshot = item.channelId, durationSecSnapshot = item.durationSec,
            syncStatus = SyncStatus.SYNCED
        )
        likedItemDao.insert(entity)
    } catch (e: Exception) {
        Timber.e(e, "Failed to add local Holodex video like")
    }
}

private fun LocalFavoriteEntity.toUnifiedDisplayItem(isDownloaded: Boolean): UnifiedDispla
    return UnifiedDisplayItem(
        stableId = "local_fav_${this.itemId}",
        playbackItemId = this.itemId,
        videoId = this.videoId,
        channelId = this.channelId,
        title = this.title,
        artistText = this.artistText,
        artworkUrls = listOfNotNull(this.artworkUrl),
        durationText = formatDurationSeconds(this.durationSec),
        isSegment = this.isSegment,
```

```
            songCount = null,
            isDownloaded = isDownloaded,
            isLiked = true,
            itemTypeForPlaylist = if (this.isSegment) LikedItemType.SONG_SEGMENT else LikedIte
            songStartSec = this.songStartSec,
            songEndSec = this.songEndSec,
            originalArtist = this.artistText,
            isExternal = true
        )
    }
}

// File: java\com\example\holodex\viewmodel\FullListViewModel.kt
// File: java/com/example/holodex/viewmodel/FullListViewModel.kt

package com.example.holodex.viewmodel

import androidx.lifecycle.SavedStateHandle
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import androidx.media3.common.util.UnstableApi
import com.example.holodex.data.model.discovery.DiscoveryChannel
import com.example.holodex.data.repository.DownloadRepository
import com.example.holodex.data.repository.HolodexRepository
import com.example.holodex.viewmodel.VideoListViewModel.ListStateHolder
import com.example.holodex.viewmodel.VideoListViewModel.MusicCategoryType
import com.example.holodex.viewmodel.mappers.toUnifiedDisplayItem
import com.example.holodex.viewmodel.mappers.toVideoShell
import dagger.hilt.android.lifecycle.HiltViewModel
import kotlinx.coroutines.flow.first
import kotlinx.coroutines.launch
import timber.log.Timber
import java.net.URLDecoder
import java.nio.charset.StandardCharsets
import javax.inject.Inject


@UnstableApi
@HiltViewModel
class FullListViewModel
@Inject constructor(
    private val savedStateHandle: SavedStateHandle,
    private val holodexRepository: HolodexRepository,
    private val downloadRepository: DownloadRepository
) : ViewModel() {

    companion object {
        const val CATEGORY_TYPE_ARG = "category"
        const val ORG_ARG = "org"
        private const val TAG = "FullListViewModel"
        private const val PAGE_SIZE = 50
    }

    val categoryType: MusicCategoryType = MusicCategoryType.valueOf(
        savedStateHandle.get<String>(CATEGORY_TYPE_ARG) ?: MusicCategoryType.TRENDING.name
```

```
)
private val organization: String = URLDecoder.decode(
    savedStateHandle.get<String>(ORG_ARG) ?: "All Vtubers",
    StandardCharsets.UTF_8.toString()
)

val listState = ListStateHolder<Any>()

init {
    Timber.d("$TAG: Initialized for category: $categoryType, org: $organization")
    loadMore(isInitialLoad = true)
}

fun loadMore(isInitialLoad: Boolean = false) {
    if (listState.isLoadingMore.value || listState.endOfList.value) return

    listState.job?.cancel()
    listState.job = viewModelScope.launch {
        if (isInitialLoad) {
            listState.isLoadingInitial.value = true
            listState.currentOffset = 0
            listState.items.value = emptyList()
        } else {
            listState.isLoadingMore.value = true
        }

        val result: Result<Any> = when (categoryType) {
            MusicCategoryType.TRENDING -> holodexRepository.getHotSongsForCarousel(organiz

            MusicCategoryType.UPCOMING_MUSIC -> holodexRepository.getUpcomingMusicPaginate
                org = organization.takeIf { it != "All Vtubers" },
                offset = listState.currentOffset
            )
            MusicCategoryType.RECENT_STREAMS -> holodexRepository.getLatestSongsPaginated(

            // --- START OF IMPLEMENTATION ---
            MusicCategoryType.COMMUNITY_PLAYLISTS -> holodexRepository.getOrgPlaylistsPagi
                org = organization,
                type = "ugp", // User Generated Playlist
                offset = listState.currentOffset,
                limit = PAGE_SIZE
            )
            MusicCategoryType.ARTIST_RADIOS -> holodexRepository.getOrgPlaylistsPaginated(
                org = organization,
                type = "radio",
                offset = listState.currentOffset,
                limit = PAGE_SIZE
            )
            MusicCategoryType.SYSTEM_PLAYLISTS -> holodexRepository.getOrgPlaylistsPaginat
                org = organization,
                type = "sgp", // System Generated Playlist
                offset = listState.currentOffset,
                limit = PAGE_SIZE
            )
            // --- END OF IMPLEMENTATION ---
```

```
        MusicCategoryType.DISCOVER_CHANNELS -> holodexRepository.getOrgChannelsPaginat
            org = organization,
            offset = listState.currentOffset,
            limit = PAGE_SIZE
        )

        else -> Result.failure(NotImplementedError("Category $categoryType not impleme
    }


    result.onSuccess { response ->
        val likedIds = holodexRepository.likedItemIds.first()
        val downloadedIds = downloadRepository.getAllDownloads().first().map { it.vide

        val newItems: List<Any> = when (response) {
            is List<*> -> {
                @Suppress("UNCHECKED_CAST")
                (response as List<com.example.holodex.data.model.discovery.MusicdexSon
                    val videoShell = song.toVideoShell()
                    song.toUnifiedDisplayItem(
                        parentVideo = videoShell,
                        isLiked = likedIds.contains("${song.videoId}_${song.start}"),
                        isDownloaded = downloadedIds.contains("${song.videoId}_${song.
                    )
                }
            }

            is com.example.holodex.data.api.PaginatedSongsResponse -> {
                response.items.map { song ->
                    val videoShell = song.toVideoShell()
                    song.toUnifiedDisplayItem(
                        parentVideo = videoShell,
                        isLiked = likedIds.contains("${song.videoId}_${song.start}"),
                        isDownloaded = downloadedIds.contains("${song.videoId}_${song.
                    )
                }
            }

            is com.example.holodex.data.api.PlaylistListResponse -> {
                response.items
            }

            is com.example.holodex.data.api.PaginatedChannelsResponse -> {
                response.items.map { it.toDiscoveryChannel() }
            }

            else -> emptyList()
        }

        if (categoryType == MusicCategoryType.DISCOVER_CHANNELS) {
            val currentChannels = if (isInitialLoad) {
                emptyList()
            } else {
                listState.items.value.filterIsInstance<DiscoveryChannel>()
```

```
                    }

                    val newChannels = newItems.filterIsInstance<DiscoveryChannel>()

                    val allChannels = (currentChannels + newChannels).distinctBy { it.id }

                    val groupedChannels = allChannels.groupBy { channel ->
                        channel.suborg?.takeIf { it.isNotBlank() } ?: organization
                    }

                    val flattenedList = mutableListOf<Any>()
                    groupedChannels.keys.sorted().forEach { subOrgName ->
                        flattenedList.add(SubOrgHeader(name = subOrgName))
                        val channelsInGroup = groupedChannels[subOrgName]
                        if (channelsInGroup != null) {
                            flattenedList.addAll(channelsInGroup.sortedBy { it.name })
                        }
                    }
                    listState.items.value = flattenedList
                } else {
                    listState.items.value += newItems
                }

                val newItemsCount = when (response) {
                    is List<*> -> response.size
                    is com.example.holodex.data.api.PaginatedSongsResponse -> response.items.s
                    is com.example.holodex.data.api.PlaylistListResponse -> response.items.siz
                    is com.example.holodex.data.api.PaginatedChannelsResponse -> response.item
                    else -> 0
                }
                listState.currentOffset += newItemsCount

                val totalAvailable = when (response) {
                    is com.example.holodex.data.api.PaginatedSongsResponse -> response.getTota
                    is com.example.holodex.data.api.PlaylistListResponse -> response.total
                    is com.example.holodex.data.api.PaginatedChannelsResponse -> response.getT
                    else -> null
                }

                if (newItemsCount < PAGE_SIZE || newItemsCount == 0 || (totalAvailable != null
                    listState.endOfList.value = true
                }
                if (categoryType == MusicCategoryType.TRENDING) {
                    listState.endOfList.value = true
                }

            }.onFailure { Timber.e(it) }

            if (isInitialLoad) listState.isLoadingInitial.value =
                false else listState.isLoadingMore.value = false
        }
    }
}

private fun com.example.holodex.data.model.discovery.ChannelDetails.toDiscoveryChannel(): Disc
```

```
    return DiscoveryChannel(
        id = this.id,
        name = this.name,
        englishName = this.englishName,
        photoUrl = this.photoUrl,
        songCount = null,
        suborg = this.group
    )
}

// File: java\com\example\holodex\viewmodel\FullPlayerViewModel.kt
// Create this new file: java/com/example/holodex/viewmodel/FullPlayerViewModel.kt
package com.example.holodex.viewmodel

import androidx.compose.ui.graphics.Color
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.example.holodex.util.DynamicTheme
import com.example.holodex.util.PaletteExtractor
import dagger.hilt.android.lifecycle.HiltViewModel
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.launch
import javax.inject.Inject

@HiltViewModel
class FullPlayerViewModel @Inject constructor(
    private val paletteExtractor: PaletteExtractor
) : ViewModel() {

    private val _dynamicTheme = MutableStateFlow(DynamicTheme.default(Color.Black, Color.White
    val dynamicTheme: StateFlow<DynamicTheme> = _dynamicTheme.asStateFlow()

    fun updateThemeFromArtwork(artworkUri: String?) {
        viewModelScope.launch {
            val defaultTheme = DynamicTheme.default(
                defaultPrimary = _dynamicTheme.value.primary,
                defaultOnPrimary = _dynamicTheme.value.onPrimary
            )
            _dynamicTheme.value = paletteExtractor.extractThemeFromUrl(artworkUri, defaultThem
        }
    }
}

// File: java\com\example\holodex\viewmodel\HistoryViewModel.kt
// File: java/com/example/holodex/viewmodel/HistoryViewModel.kt

package com.example.holodex.viewmodel

import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import androidx.media3.common.util.UnstableApi
import com.example.holodex.data.repository.DownloadRepository
import com.example.holodex.data.repository.HolodexRepository
```

```
import com.example.holodex.playback.PlaybackRequestManager
import com.example.holodex.playback.domain.usecase.AddItemsToQueueUseCase
import com.example.holodex.viewmodel.mappers.toPlaybackItem
import com.example.holodex.viewmodel.mappers.toUnifiedDisplayItem
import dagger.hilt.android.lifecycle.HiltViewModel
import kotlinx.coroutines.flow.MutableSharedFlow
import kotlinx.coroutines.flow.SharedFlow
import kotlinx.coroutines.flow.SharingStarted
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asSharedFlow
import kotlinx.coroutines.flow.catch
import kotlinx.coroutines.flow.combine
import kotlinx.coroutines.flow.map
import kotlinx.coroutines.flow.stateIn
import kotlinx.coroutines.launch
import timber.log.Timber
import javax.inject.Inject

@UnstableApi
@HiltViewModel
class HistoryViewModel @Inject constructor(
    private val holodexRepository: HolodexRepository,
    private val downloadRepository: DownloadRepository,
    private val playbackRequestManager: PlaybackRequestManager,
    private val addItemsToQueueUseCase: AddItemsToQueueUseCase
) : ViewModel() {

    private val _transientMessage = MutableSharedFlow<String>()
    val transientMessage: SharedFlow<String> = _transientMessage.asSharedFlow()

    // --- REVERT TO THE CORRECT THREE-WAY COMBINE PATTERN ---
    val unifiedHistoryItems: StateFlow<List<UnifiedDisplayItem>> =
        combine(
            holodexRepository.getHistory(),
            holodexRepository.likedItemIds, // This is now a reliable StateFlow
            downloadRepository.getAllDownloads().map { list -> list.map { it.videoId }.toSet()
        ) { historyEntities, likedIds, downloadedIds ->
            // This transformation will now run automatically whenever history, likes, or down
            historyEntities.map { entity ->
                entity.toUnifiedDisplayItem(
                    isDownloaded = downloadedIds.contains(entity.itemId),
                    isLiked = likedIds.contains(entity.itemId)
                )
            }
        }
            .catch { e ->
                Timber.e(e, "Error combining history items.")
                emit(emptyList())
            }
            .stateIn(
                scope = viewModelScope,
                started = SharingStarted.WhileSubscribed(5000L),
                initialValue = emptyList()
            )
    // --- END OF CORRECTION ---
```

```
    fun playFromHistoryItem(tappedItem: UnifiedDisplayItem) {
        viewModelScope.launch {
            val currentHistory = unifiedHistoryItems.value
            val tappedIndex = currentHistory.indexOf(tappedItem)

            if (tappedIndex == -1) {
                _transientMessage.emit("Error: Could not find item to play.")
                return@launch
            }

            val playbackItems = currentHistory.map { it.toPlaybackItem() }
            playbackRequestManager.submitPlaybackRequest(
                items = playbackItems,
                startIndex = tappedIndex
            )
        }
    }

    fun playAllHistory() {
        viewModelScope.launch {
            val playbackItems = unifiedHistoryItems.value.map { it.toPlaybackItem() }
            if (playbackItems.isNotEmpty()) {
                playbackRequestManager.submitPlaybackRequest(items = playbackItems)
            } else {
                _transientMessage.emit("History is empty.")
            }
        }
    }

    fun addAllHistoryToQueue() {
        viewModelScope.launch {
            val playbackItems = unifiedHistoryItems.value.map { it.toPlaybackItem() }
            if (playbackItems.isNotEmpty()) {
                addItemsToQueueUseCase(playbackItems)
                _transientMessage.emit("Added ${playbackItems.size} songs to queue.")
            } else {
                _transientMessage.emit("History is empty.")
            }
        }
    }
}

// File: java\com\example\holodex\viewmodel\PlaybackUiStateSelectors.kt
// File: java/com/example/holodex/viewmodel/PlaybackUiStateSelectors.kt
package com.example.holodex.viewmodel

import androidx.compose.runtime.Composable
import androidx.compose.runtime.State
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.hilt.navigation.compose.hiltViewModel
import androidx.lifecycle.compose.collectAsStateWithLifecycle
```

```
import com.example.holodex.playback.domain.model.DomainPlaybackProgress
import com.example.holodex.playback.domain.model.DomainRepeatMode
import com.example.holodex.playback.domain.model.DomainShuffleMode
import com.example.holodex.playback.domain.model.PlaybackItem
import com.example.holodex.util.getHighResArtworkUrl
import kotlinx.coroutines.flow.StateFlow

// --- Selectors for MiniPlayer ---

@Composable
fun rememberMiniPlayerArtworkState(
    uiStateFlow: StateFlow<PlaybackUiState>,
    settingsViewModel: SettingsViewModel = hiltViewModel()
): State<String?> {
    val uiState by uiStateFlow.collectAsStateWithLifecycle()
    val imageQuality by settingsViewModel.currentImageQuality.collectAsStateWithLifecycle()

    return remember(uiState.currentItem?.artworkUri, imageQuality) {
        mutableStateOf(
            getHighResArtworkUrl(
                uiState.currentItem?.artworkUri,
                imageQualityKey = imageQuality,
                preferredSizeOverride = "200x200"
            )
        )
    }
}

@Composable
fun rememberMiniPlayerTitleState(
    uiStateFlow: StateFlow<PlaybackUiState>
): State<String?> {
    val uiState by uiStateFlow.collectAsStateWithLifecycle()
    return remember(uiState.currentItem?.id) { mutableStateOf(uiState.currentItem?.title) }
}

@Composable
fun rememberMiniPlayerArtistState(
    uiStateFlow: StateFlow<PlaybackUiState>
): State<String?> {
    val uiState by uiStateFlow.collectAsStateWithLifecycle()
    return remember(uiState.currentItem?.id) { mutableStateOf(uiState.currentItem?.artistText)
}

@Composable
fun rememberIsPlayingState(
    uiStateFlow: StateFlow<PlaybackUiState>
): State<Boolean> {
    val uiState by uiStateFlow.collectAsStateWithLifecycle()
    return remember(uiState.isPlaying) { mutableStateOf(uiState.isPlaying) }
}

@Composable
fun rememberMiniPlayerProgressState(
    uiStateFlow: StateFlow<PlaybackUiState>
```

```
): State<Float> {
    val uiState by uiStateFlow.collectAsStateWithLifecycle()
    return remember(uiState.progress, uiState.currentItem?.id) {
        val progressFraction = if (uiState.currentItem != null && uiState.progress.durationSec
            (uiState.progress.positionSec.toFloat() / uiState.progress.durationSec.toFloat()).
        } else {
            0f
        }
        mutableStateOf(progressFraction)
    }
}

@Composable
fun rememberMiniPlayerQueueStateForButton(
    uiStateFlow: StateFlow<PlaybackUiState>
): State<Pair<Boolean, Boolean>> {
    val uiState by uiStateFlow.collectAsStateWithLifecycle()
    return remember(uiState.queue, uiState.currentItem, uiState.currentIndexInQueue, uiState.r
        val hasItemAndQueue = uiState.queue.isNotEmpty() && uiState.currentItem != null
        val canSkipNext = hasItemAndQueue &&
                (uiState.currentIndexInQueue < uiState.queue.size - 1 || uiState.queue.size ==
        mutableStateOf(hasItemAndQueue to canSkipNext)
    }
}


// --- Selectors for FullPlayerScreen ---

@Composable
fun rememberFullPlayerArtworkState(
    uiStateFlow: StateFlow<PlaybackUiState>,
    settingsViewModel: SettingsViewModel = hiltViewModel()
): State<String?> {
    val uiState by uiStateFlow.collectAsStateWithLifecycle()
    val imageQuality by settingsViewModel.currentImageQuality.collectAsStateWithLifecycle()

    return remember(uiState.currentItem?.artworkUri, imageQuality) {
        mutableStateOf(
            getHighResArtworkUrl(
                uiState.currentItem?.artworkUri,
                imageQualityKey = imageQuality,
                preferredSizeOverride = null
            )
        )
    }
}

@Composable
fun rememberFullPlayerCurrentItemState(
    uiStateFlow: StateFlow<PlaybackUiState>
): State<PlaybackItem?> {
    val uiState by uiStateFlow.collectAsStateWithLifecycle()
    return remember(uiState.currentItem) { mutableStateOf(uiState.currentItem) }
}
```

```
@Composable
fun rememberFullPlayerProgressState(
    uiStateFlow: StateFlow<PlaybackUiState>
): State<DomainPlaybackProgress> {
    val uiState by uiStateFlow.collectAsStateWithLifecycle()
    return remember(uiState.progress) { mutableStateOf(uiState.progress) }
}

@Composable
fun rememberFullPlayerQueueInfoState(
    uiStateFlow: StateFlow<PlaybackUiState>
): State<Triple<List<PlaybackItem>, Int, Boolean>> {
    val uiState by uiStateFlow.collectAsStateWithLifecycle()
    return remember(uiState.queue, uiState.currentIndexInQueue) {
        mutableStateOf(Triple(uiState.queue, uiState.currentIndexInQueue, uiState.queue.isNotE
    }
}

@Composable
fun rememberFullPlayerLoadingState(
    uiStateFlow: StateFlow<PlaybackUiState>
): State<Boolean> {
    val uiState by uiStateFlow.collectAsStateWithLifecycle()
    return remember(uiState.isLoading) { mutableStateOf(uiState.isLoading) }
}
@Composable
fun rememberFullPlayerControlModesState(
    uiStateFlow: StateFlow<PlaybackUiState>
): State<Pair<DomainRepeatMode, DomainShuffleMode>> {
    val uiState by uiStateFlow.collectAsStateWithLifecycle()
    return remember(uiState.repeatMode, uiState.shuffleMode) {
        mutableStateOf(uiState.repeatMode to uiState.shuffleMode)
    }
}

// File: java\com\example\holodex\viewmodel\PlaybackViewModel.kt
// File: java/com/example/holodex/viewmodel/PlaybackViewModel.kt
package com.example.holodex.viewmodel

import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.example.holodex.playback.domain.model.DomainPlaybackProgress
import com.example.holodex.playback.domain.model.DomainPlaybackState
import com.example.holodex.playback.domain.model.DomainRepeatMode
import com.example.holodex.playback.domain.model.DomainShuffleMode
import com.example.holodex.playback.domain.model.PlaybackItem
import com.example.holodex.playback.domain.repository.PlaybackRepository
import com.example.holodex.playback.domain.usecase.AddItemToQueueUseCase
import com.example.holodex.playback.domain.usecase.AddItemsToQueueUseCase
import com.example.holodex.playback.domain.usecase.ClearQueueUseCase
import com.example.holodex.playback.domain.usecase.GetPlayerSessionIdUseCase
import com.example.holodex.playback.domain.usecase.ObserveCurrentPlayingItemUseCase
import com.example.holodex.playback.domain.usecase.ObservePlaybackProgressUseCase
import com.example.holodex.playback.domain.usecase.ObservePlaybackQueueUseCase
import com.example.holodex.playback.domain.usecase.ObservePlaybackStateUseCase
```

```
import com.example.holodex.playback.domain.usecase.PausePlaybackUseCase
import com.example.holodex.playback.domain.usecase.PlayItemsUseCase
import com.example.holodex.playback.domain.usecase.RemoveItemFromQueueUseCase
import com.example.holodex.playback.domain.usecase.ReorderQueueItemUseCase
import com.example.holodex.playback.domain.usecase.ResumePlaybackUseCase
import com.example.holodex.playback.domain.usecase.SeekPlaybackUseCase
import com.example.holodex.playback.domain.usecase.SetRepeatModeUseCase
import com.example.holodex.playback.domain.usecase.SetScrubbingUseCase
import com.example.holodex.playback.domain.usecase.SetShuffleModeUseCase
import com.example.holodex.playback.domain.usecase.SkipToNextItemUseCase
import com.example.holodex.playback.domain.usecase.SkipToPreviousItemUseCase
import com.example.holodex.playback.domain.usecase.SkipToQueueItemUseCase
import com.example.holodex.viewmodel.autoplay.ContinuationManager
import dagger.hilt.android.lifecycle.HiltViewModel
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.SharingStarted
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.flow.combine
import kotlinx.coroutines.flow.first
import kotlinx.coroutines.flow.stateIn
import kotlinx.coroutines.launch
import timber.log.Timber
import javax.inject.Inject

data class PlaybackUiState(
    val currentItem: PlaybackItem? = null,
    val isPlaying: Boolean = false,
    val progress: DomainPlaybackProgress = DomainPlaybackProgress.NONE,
    val queue: List<PlaybackItem> = emptyList(),
    val currentIndexInQueue: Int = -1,
    val repeatMode: DomainRepeatMode = DomainRepeatMode.NONE,
    val shuffleMode: DomainShuffleMode = DomainShuffleMode.OFF,
    val isLoading: Boolean = false
)

@HiltViewModel
class PlaybackViewModel @Inject constructor(
    private val playItemsUseCase: PlayItemsUseCase,
    private val pausePlaybackUseCase: PausePlaybackUseCase,
    private val resumePlaybackUseCase: ResumePlaybackUseCase,
    private val seekPlaybackUseCase: SeekPlaybackUseCase,
    private val skipToNextItemUseCase: SkipToNextItemUseCase,
    private val skipToPreviousItemUseCase: SkipToPreviousItemUseCase,
    private val setRepeatModeUseCase: SetRepeatModeUseCase,
    private val setShuffleModeUseCase: SetShuffleModeUseCase,
    observeCurrentPlayingItemUseCase: ObserveCurrentPlayingItemUseCase,
    observePlaybackStateUseCase: ObservePlaybackStateUseCase,
    continuationManager: ContinuationManager,
    observePlaybackProgressUseCase: ObservePlaybackProgressUseCase,
    observePlaybackQueueUseCase: ObservePlaybackQueueUseCase,
    private val setScrubbingUseCase: SetScrubbingUseCase,
    private val addItemToQueueUseCase: AddItemToQueueUseCase,
    private val addItemsToQueueUseCase: AddItemsToQueueUseCase,
    private val removeItemFromQueueUseCase: RemoveItemFromQueueUseCase,
```

```
    private val reorderQueueItemUseCase: ReorderQueueItemUseCase,
    private val clearQueueUseCase: ClearQueueUseCase,
    private val skipToQueueItemUseCase: SkipToQueueItemUseCase,
    private val getPlayerSessionIdUseCase: GetPlayerSessionIdUseCase,
    private val playbackRepository: PlaybackRepository
) : ViewModel() {
    companion object {
        private const val TAG = "PlaybackViewModel"
    }
    private val _isVmPreparingPlayback = MutableStateFlow(false)

    val isRadioModeActive: StateFlow<Boolean> = continuationManager.isRadioModeActive
        .stateIn(
            scope = viewModelScope,
            started = SharingStarted.WhileSubscribed(5000L),
            initialValue = false
        )

    val uiState: StateFlow<PlaybackUiState> = combine(
        observeCurrentPlayingItemUseCase(),
        observePlaybackStateUseCase(),
        observePlaybackProgressUseCase(),
        observePlaybackQueueUseCase(),
        _isVmPreparingPlayback.asStateFlow()
    ) { currentItem, domainPlayerState, progress, queueData, vmIsPreparing ->
        val isActuallyPlaying = domainPlayerState == DomainPlaybackState.PLAYING
        val isBufferingFromPlayer = domainPlayerState == DomainPlaybackState.BUFFERING
        PlaybackUiState(
            currentItem,
            isActuallyPlaying,
            progress,
            queueData.items,
            queueData.currentIndex,
            queueData.repeatMode,
            queueData.shuffleMode,
            isBufferingFromPlayer || vmIsPreparing
        )
    }.stateIn(
        scope = viewModelScope,
        started = SharingStarted.WhileSubscribed(5000L),
        initialValue = PlaybackUiState(isLoading = true)
    )

    init {
        Timber.d("$TAG initialized.")
        viewModelScope.launch {
            uiState.first()
            _isVmPreparingPlayback.value = false
        }
    }

    fun playItems(items: List<PlaybackItem>, startIndex: Int = 0, startPositionMs: Long = 0L,
        viewModelScope.launch {
            if (items.isEmpty()) return@launch
            _isVmPreparingPlayback.value = true
```

```
        try {
            val effectiveShuffle = shouldShuffle ?: (uiState.value.shuffleMode == DomainSh
            playItemsUseCase(items, startIndex, startPositionMs, effectiveShuffle)
        } finally {
            _isVmPreparingPlayback.value = false
        }
    }
}

fun togglePlayPause() {
    viewModelScope.launch {
        if (uiState.value.isPlaying) {
            pausePlaybackUseCase()
        } else if (uiState.value.currentItem == null && uiState.value.queue.isNotEmpty())
            val startIndex = uiState.value.currentIndexInQueue.coerceAtLeast(0)
            val startPosMs = if (uiState.value.currentIndexInQueue == startIndex) uiState.
            playItems(uiState.value.queue, startIndex, startPosMs)
        } else {
            resumePlaybackUseCase()
        }
    }
}

fun seekTo(positionSec: Long) = viewModelScope.launch { seekPlaybackUseCase(positionSec) }
fun skipToNext() = viewModelScope.launch { skipToNextItemUseCase() }
fun skipToPrevious() = viewModelScope.launch { skipToPreviousItemUseCase() }
fun setScrubbing(isScrubbing: Boolean) = viewModelScope.launch { setScrubbingUseCase(isScr

fun toggleRepeatMode() = viewModelScope.launch {
    val nextMode = when (uiState.value.repeatMode) {
        DomainRepeatMode.NONE -> DomainRepeatMode.ALL
        DomainRepeatMode.ALL -> DomainRepeatMode.ONE
        DomainRepeatMode.ONE -> DomainRepeatMode.NONE
    }
    setRepeatModeUseCase(nextMode)
}

fun toggleShuffleMode() = viewModelScope.launch {
    val nextMode = if (uiState.value.shuffleMode == DomainShuffleMode.ON) DomainShuffleMod
    setShuffleModeUseCase(nextMode)
}

fun playQueueItemAtIndex(index: Int) {
    viewModelScope.launch {
        if (index in uiState.value.queue.indices) {
            skipToQueueItemUseCase(index)
        }
    }
}

fun addItemToQueue(item: PlaybackItem, index: Int? = null) = viewModelScope.launch { addIt
fun addItemsToQueue(items: List<PlaybackItem>, index: Int? = null) = viewModelScope.launch
fun removeItemFromQueue(index: Int) = viewModelScope.launch { removeItemFromQueueUseCase(i
fun reorderQueueItem(fromIndex: Int, toIndex: Int) = viewModelScope.launch { reorderQueueI
fun clearCurrentQueue() = viewModelScope.launch { clearQueueUseCase() }
```

```
    fun getAudioSessionId(): Int? = getPlayerSessionIdUseCase()
}

// File: java\com\example\holodex\viewmodel\PlaylistDetailsViewModel.kt
// File: java/com/example/holodex/viewmodel/PlaylistDetailsViewModel.kt
package com.example.holodex.viewmodel

import android.app.Application
import androidx.compose.ui.graphics.Color
import androidx.lifecycle.SavedStateHandle
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import androidx.media3.common.util.UnstableApi
import com.example.holodex.R
import com.example.holodex.auth.TokenManager
import com.example.holodex.data.db.DownloadedItemEntity
import com.example.holodex.data.db.LikedItemEntity
import com.example.holodex.data.db.PlaylistEntity
import com.example.holodex.data.db.PlaylistItemEntity
import com.example.holodex.data.db.SyncStatus
import com.example.holodex.data.model.discovery.MusicdexSong
import com.example.holodex.data.model.discovery.PlaylistStub
import com.example.holodex.data.repository.DownloadRepository
import com.example.holodex.data.repository.HolodexRepository
import com.example.holodex.playback.PlaybackRequestManager
import com.example.holodex.playback.domain.repository.PlaybackRepository
import com.example.holodex.playback.domain.usecase.AddItemsToQueueUseCase
import com.example.holodex.util.ArtworkResolver
import com.example.holodex.util.DynamicTheme
import com.example.holodex.util.PaletteExtractor
import com.example.holodex.util.PlaylistFormatter
import com.example.holodex.viewmodel.mappers.toPlaybackItem
import com.example.holodex.viewmodel.mappers.toUnifiedDisplayItem
import com.example.holodex.viewmodel.mappers.toVideoShell
import dagger.hilt.android.lifecycle.HiltViewModel
import kotlinx.coroutines.flow.MutableSharedFlow
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.SharedFlow
import kotlinx.coroutines.flow.SharingStarted
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asSharedFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.flow.combine
import kotlinx.coroutines.flow.first
import kotlinx.coroutines.flow.map
import kotlinx.coroutines.flow.stateIn
import kotlinx.coroutines.flow.update
import kotlinx.coroutines.launch
import timber.log.Timber
import java.time.Instant
import javax.inject.Inject

@UnstableApi
@HiltViewModel
class PlaylistDetailsViewModel
```

```
@Inject constructor(
    private val application: Application,
    savedStateHandle: SavedStateHandle,
    private val holodexRepository: HolodexRepository,
    private val downloadRepository: DownloadRepository,
    private val playbackRequestManager: PlaybackRequestManager,
    private val playbackRepository: PlaybackRepository,
    private val addItemsToQueueUseCase: AddItemsToQueueUseCase,
    private val paletteExtractor: PaletteExtractor,
    private val tokenManager: TokenManager
) : ViewModel() {

    companion object {
        const val PLAYLIST_ID_ARG = "playlistId"
        const val LIKED_SEGMENTS_PLAYLIST_ID = "-100"
        const val DOWNLOADS_PLAYLIST_ID = "-200"
    }

    val playlistId: String = savedStateHandle.get<String>(PLAYLIST_ID_ARG) ?: ""

    private val _playlistDetails = MutableStateFlow<PlaylistEntity?>(null)
    val playlistDetails: StateFlow<PlaylistEntity?> = _playlistDetails.asStateFlow()

    private val _isLoading = MutableStateFlow(true)
    val isLoading: StateFlow<Boolean> = _isLoading.asStateFlow()

    private val _error = MutableStateFlow<String?>(null)
    val error: StateFlow<String?> = _error.asStateFlow()

    private val _isPlaylistShuffleActive = MutableStateFlow(false)
    val isPlaylistShuffleActive: StateFlow<Boolean> = _isPlaylistShuffleActive.asStateFlow()

    private val _rawItemsFlow = MutableStateFlow<List<Any>>(emptyList())

    private val _transientMessage = MutableSharedFlow<String>()
    val transientMessage: SharedFlow<String> = _transientMessage.asSharedFlow()

    private val _dynamicTheme = MutableStateFlow(DynamicTheme.default(Color.Black, Color.White
    val dynamicTheme: StateFlow<DynamicTheme> = _dynamicTheme.asStateFlow()

    private val _isEditMode = MutableStateFlow(false)
    val isEditMode: StateFlow<Boolean> = _isEditMode.asStateFlow()

    private val _editablePlaylist = MutableStateFlow<PlaylistEntity?>(null)
    val editablePlaylist: StateFlow<PlaylistEntity?> = _editablePlaylist.asStateFlow()

    private val _editableItems = MutableStateFlow<List<PlaylistItemEntity>>(emptyList())

    val isPlaylistOwned: StateFlow<Boolean> = _playlistDetails.map { playlist ->
        playlist?.owner != null && playlist.owner.toString() == tokenManager.getUserId()
    }.stateIn(viewModelScope, SharingStarted.WhileSubscribed(5000L), false)

    val unifiedPlaylistItems: StateFlow<List<UnifiedDisplayItem>> = combine(
        _isEditMode, _rawItemsFlow, _editableItems,
        holodexRepository.likedItemIds,
```

```
    downloadRepository.getAllDownloads().map { list -> list.map { it.videoId }.toSet() }
) { isEditing, rawItems, editableItems, likedIds, downloadedIds ->
    val itemsToDisplay = if (isEditing) editableItems else rawItems
    itemsToDisplay.mapNotNull { rawItem ->
        when (rawItem) {
            is PlaylistItemEntity -> rawItem.toUnifiedDisplayItem(
                isDownloaded = downloadedIds.contains(rawItem.itemIdInPlaylist),
                isLiked = likedIds.contains(rawItem.itemIdInPlaylist)
            )
            is LikedItemEntity -> rawItem.toUnifiedDisplayItem(
                isDownloaded = downloadedIds.contains(rawItem.itemId)
            )
            is DownloadedItemEntity -> rawItem.toUnifiedDisplayItem(
                isLiked = likedIds.contains(rawItem.videoId)
            )
            is MusicdexSong -> {
                val videoShell = rawItem.toVideoShell(_playlistDetails.value?.name ?: "")
                rawItem.toUnifiedDisplayItem(
                    parentVideo = videoShell,
                    isLiked = likedIds.contains("${rawItem.videoId}_${rawItem.start}"),
                    isDownloaded = downloadedIds.contains("${rawItem.videoId}_${rawItem.st
                )
            }
            else -> null
        }
    }
}.stateIn(viewModelScope, SharingStarted.WhileSubscribed(5000L), emptyList())

fun togglePlaylistShuffleMode() {
    _isPlaylistShuffleActive.value = !_isPlaylistShuffleActive.value
}

init {
    if (playlistId.isNotBlank()) {
        loadPlaylistDetails()
    } else {
        _isLoading.value = false
        _error.value = "Invalid Playlist ID."
    }
}

fun loadPlaylistDetails() {
    viewModelScope.launch {
        _isLoading.value = true
        _error.value = null
        val now = Instant.now().toString()
        var artworkUrl: String? = null

        try {
            when (playlistId) {
                LIKED_SEGMENTS_PLAYLIST_ID -> {
                    val likedItems = holodexRepository.getObservableLikedSongSegments().fi
                    _playlistDetails.value = PlaylistEntity(
                        playlistId = LIKED_SEGMENTS_PLAYLIST_ID.toLong(),
                        name = application.getString(R.string.playlist_title_liked_segment
```

```
                    description = application.getString(R.string.playlist_desc_liked_s
                    createdAt = now, last_modified_at = now, serverId = null, owner =
                )
                _rawItemsFlow.value = likedItems
                artworkUrl = likedItems.firstOrNull()?.artworkUrlSnapshot
            }

            DOWNLOADS_PLAYLIST_ID -> {
                val downloadedItems = downloadRepository.getAllDownloads().first()
                _playlistDetails.value = PlaylistEntity(
                    playlistId = DOWNLOADS_PLAYLIST_ID.toLong(),
                    name = application.getString(R.string.playlist_title_downloads),
                    description = application.getString(R.string.playlist_desc_downloa
                    createdAt = now, last_modified_at = now, serverId = null, owner =
                )
                _rawItemsFlow.value = downloadedItems
                artworkUrl = downloadedItems.firstOrNull()?.artworkUrl
            }

            else -> {
                val longId = playlistId.toLongOrNull()
                if (longId != null && longId > 0) {
                    val playlist = holodexRepository.getPlaylistById(longId)
                    val items = holodexRepository.getItemsForPlaylist(longId).first()
                    _playlistDetails.value = playlist
                    _rawItemsFlow.value = items
                    artworkUrl = items.firstOrNull()?.songArtworkUrlPlaylist
                } else {
                    val isRadio = playlistId.startsWith(":artist") || playlistId.start
                    val result = if (isRadio) holodexRepository.getRadioContent(playli

                    result.onSuccess { fullPlaylist ->
                        val tempStub = PlaylistStub(
                            id = fullPlaylist.id, title = fullPlaylist.title, type = f
                            description = fullPlaylist.description, artContext = null
                        )
                        val formattedTitle = PlaylistFormatter.getDisplayTitle(tempStu
                        val formattedDescription = PlaylistFormatter.getDisplayDescrip

                        _playlistDetails.value = PlaylistEntity(
                            playlistId = 0L, name = formattedTitle, description = form
                            createdAt = fullPlaylist.createdAt, last_modified_at = ful
                            serverId = fullPlaylist.id, owner = null
                        )
                        _rawItemsFlow.value = fullPlaylist.content ?: emptyList()
                        artworkUrl = ArtworkResolver.getPlaylistArtworkUrl(tempStub) ?
                    }.onFailure { throw it }
                }
            }
        }
        _dynamicTheme.value = paletteExtractor.extractThemeFromUrl(
            artworkUrl,
            DynamicTheme.default(Color.Black, Color.White)
        )
    } catch (e: Exception) {
```

```
            Timber.e(e, "Failed to load details for playlist ID: $playlistId")
            _error.value = "Failed to load playlist: ${e.localizedMessage}"
        } finally {
            _isLoading.value = false
        }
    }
}

fun playAllItemsInPlaylist() {
    viewModelScope.launch {
        val isRadio = playlistId.startsWith(":")
        if (isRadio) {
            playbackRepository.prepareAndPlayRadio(playlistId)
        } else {
            val itemsToPlay = unifiedPlaylistItems.value
            if (itemsToPlay.isEmpty()) {
                _error.value = "Playlist is empty."
                return@launch
            }
            val isShuffleOn = _isPlaylistShuffleActive.value
            val playbackItems = itemsToPlay.map { it.toPlaybackItem() }
            playbackRequestManager.submitPlaybackRequest(
                items = playbackItems, startIndex = 0, shouldShuffle = isShuffleOn
            )
        }
    }
}

fun addAllToQueue() {
    viewModelScope.launch {
        val items = unifiedPlaylistItems.value
        if (items.isNotEmpty()) {
            val playbackItems = items.map { it.toPlaybackItem() }
            addItemsToQueueUseCase(playbackItems)
            _transientMessage.emit("Added ${items.size} songs to the queue")
        }
    }
}

fun playFromItem(tappedItem: UnifiedDisplayItem) {
    viewModelScope.launch {
        val allItems = unifiedPlaylistItems.value
        if (allItems.isEmpty()) return@launch
        val startIndex = allItems.indexOf(tappedItem).coerceAtLeast(0)
        val playbackItems = allItems.map { it.toPlaybackItem() }
        playbackRequestManager.submitPlaybackRequest(
            playbackItems, startIndex, shouldShuffle = _isPlaylistShuffleActive.value
        )
    }
}

fun enterEditMode() {
    _playlistDetails.value?.let { originalPlaylist ->
        _editablePlaylist.value = originalPlaylist.copy()
        _editableItems.value = _rawItemsFlow.value.filterIsInstance<PlaylistItemEntity>()
```

```
            _isEditMode.value = true
        }
    }

    fun cancelEditMode() {
        _isEditMode.value = false
        _editablePlaylist.value = null
        _editableItems.value = emptyList()
    }

    fun updateDraftName(newName: String) {
        _editablePlaylist.update { it?.copy(name = newName) }
    }

    fun updateDraftDescription(newDescription: String) {
        _editablePlaylist.update { it?.copy(description = newDescription) }
    }

    fun reorderItemInEditMode(from: Int, to: Int) {
        val currentList = _editableItems.value.toMutableList()
        val movedItem = currentList.removeAt(from)
        currentList.add(to, movedItem)
        _editableItems.value = currentList
    }

    fun removeItemInEditMode(itemToRemove: UnifiedDisplayItem) {
        _editableItems.update { currentList ->
            currentList.filterNot { it.itemIdInPlaylist == itemToRemove.playbackItemId }
        }
    }

    fun saveChanges() = viewModelScope.launch {
        val originalPlaylist = _playlistDetails.value ?: return@launch
        val draftPlaylist = _editablePlaylist.value
        val draftItems = _editableItems.value

        if (draftPlaylist == null || draftPlaylist.name.isNullOrBlank()) {
            _error.value = "Playlist name cannot be empty."
            return@launch
        }

        val originalSyncedItemIds = _rawItemsFlow.value.filterIsInstance<PlaylistItemEntity>()
            .filter { !it.isLocalOnly }.map { it.itemIdInPlaylist }
        val newSyncedItemIds = draftItems.filter { !it.isLocalOnly }.map { it.itemIdInPlaylist

        val contentChanged = originalSyncedItemIds != newSyncedItemIds
        val metadataChanged = originalPlaylist.name != draftPlaylist.name || originalPlaylist.

        val finalPlaylistState = if (contentChanged || metadataChanged) {
            draftPlaylist.copy(syncStatus = SyncStatus.DIRTY, last_modified_at = Instant.now()
        } else {
            draftPlaylist
        }

        try {
```

```
            holodexRepository.savePlaylistEdits(finalPlaylistState, draftItems)
            cancelEditMode()
            loadPlaylistDetails()
        } catch (e: Exception) {
            Timber.e(e, "Failed to save playlist edits.")
            _error.value = "Failed to save changes: ${e.localizedMessage}"
        }
    }

    fun clearError() {
        _error.value = null
    }

}

// File: java\com\example\holodex\viewmodel\PlaylistManagementViewModel.kt
// File: java/com/example/holodex/viewmodel/PlaylistManagementViewModel.kt
package com.example.holodex.viewmodel

import android.app.Application
import android.widget.Toast
import androidx.annotation.OptIn
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import androidx.media3.common.util.UnstableApi
import com.example.holodex.R
import com.example.holodex.data.db.DownloadedItemEntity
import com.example.holodex.data.db.LikedItemEntity
import com.example.holodex.data.db.LikedItemType
import com.example.holodex.data.db.PlaylistEntity
import com.example.holodex.data.db.PlaylistItemEntity
import com.example.holodex.data.db.StarredPlaylistEntity
import com.example.holodex.data.model.discovery.PlaylistStub
import com.example.holodex.data.repository.DownloadRepository
import com.example.holodex.data.repository.HolodexRepository
import com.example.holodex.data.repository.LocalRepository
import com.example.holodex.playback.domain.model.PlaybackItem
import com.example.holodex.util.PlaylistFormatter
import dagger.hilt.android.lifecycle.HiltViewModel
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.SharingStarted
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.flow.combine
import kotlinx.coroutines.flow.stateIn
import kotlinx.coroutines.launch
import timber.log.Timber
import java.time.Instant
import javax.inject.Inject
import kotlin.math.absoluteValue

data class PendingPlaylistItemDetails(
    val videoId: String,
    val itemType: LikedItemType,
```

```
    val titleForDisplay: String,
    val artistForDisplay: String,
    val artworkForDisplay: String?,
    val songStartSeconds: Int? = null,
    val songEndSeconds: Int? = null,
    val isExternal: Boolean
)

@OptIn(UnstableApi::class)
@HiltViewModel
class PlaylistManagementViewModel @Inject constructor(
    private val application: Application,
    private val holodexRepository: HolodexRepository,
    private val localRepository: LocalRepository,
    private val downloadRepository: DownloadRepository
) : ViewModel() {

    companion object {
        const val TAG = "PlaylistMgmtVM"
        const val LIKED_SEGMENTS_PLAYLIST_ID = -100L
        const val DOWNLOADS_PLAYLIST_ID = -200L
    }

    private val userCreatedPlaylists: Flow<List<PlaylistEntity>> = holodexRepository.getAllPla
    private val starredPlaylists: Flow<List<StarredPlaylistEntity>> = holodexRepository.getSta
    private val downloadsFlow: Flow<List<DownloadedItemEntity>> = downloadRepository.getAllDow

    val allDisplayablePlaylists: StateFlow<List<PlaylistEntity>> =
        combine(
            userCreatedPlaylists,
            starredPlaylists,
            downloadsFlow,
            localRepository.getAllLocalPlaylists() // Combine local playlists
        ) { userPlaylists, starred, downloads, localPlaylists ->
            val syntheticPlaylists = mutableListOf<PlaylistEntity>()
            val now = Instant.now().toString()

            syntheticPlaylists.add(
                PlaylistEntity(
                    playlistId = LIKED_SEGMENTS_PLAYLIST_ID,
                    name = application.getString(R.string.playlist_title_liked_segments),
                    description = application.getString(R.string.playlist_desc_liked_segments)
                    createdAt = now, last_modified_at = now, serverId = null, owner = null
                )
            )

            if (downloads.any { it.downloadStatus == com.example.holodex.data.db.DownloadStatu
                syntheticPlaylists.add(
                    PlaylistEntity(
                        playlistId = DOWNLOADS_PLAYLIST_ID,
                        name = application.getString(R.string.playlist_title_downloads),
                        description = application.getString(R.string.playlist_desc_downloads),
                        createdAt = now, last_modified_at = now, serverId = null, owner = null
                    )
                )
```

```
        }

        val starredAsDisplayable = starred.map { starredItem ->
            val userPlaylistMatch = userPlaylists.find { it.serverId == starredItem.playli
            if (userPlaylistMatch != null) {
                userPlaylistMatch
            } else {
                val uniqueNegativeId = ("starred_${starredItem.playlistId}".hashCode()).ab
                val tempStub = PlaylistStub(
                    id = starredItem.playlistId,
                    title = "Starred Playlist", type = "", artContext = null,
                    description = "ID: ${starredItem.playlistId}"
                )
                val formattedTitle = PlaylistFormatter.getDisplayTitle(tempStub, applicati
                val formattedDescription = PlaylistFormatter.getDisplayDescription(tempStu
                PlaylistEntity(
                    playlistId = uniqueNegativeId, serverId = starredItem.playlistId, name
                    description = formattedDescription, createdAt = now, last_modified_at
                )
            }
        }

        val starredServerIds = starred.map { it.playlistId }.toSet()
        val uniqueUserPlaylists = userPlaylists.filter { it.serverId !in starredServerIds

        // Add local playlists from the new table
        val localAsDisplayable = localPlaylists.map { localPlaylist ->
            PlaylistEntity(
                playlistId = localPlaylist.localPlaylistId * -1, // Use negative ID to dis
                name = localPlaylist.name,
                description = localPlaylist.description,
                createdAt = Instant.ofEpochMilli(localPlaylist.createdAt).toString(),
                last_modified_at = Instant.ofEpochMilli(localPlaylist.createdAt).toString(
                serverId = null, owner = null
            )
        }

        syntheticPlaylists + uniqueUserPlaylists + starredAsDisplayable + localAsDisplayab
    }.stateIn(
        scope = viewModelScope,
        started = SharingStarted.WhileSubscribed(5000L),
        initialValue = emptyList()
    )

val userPlaylists: StateFlow<List<PlaylistEntity>> = holodexRepository.getAllPlaylists()
    .stateIn(
        scope = viewModelScope,
        started = SharingStarted.WhileSubscribed(5000L),
        initialValue = emptyList()
    )

private val _showCreatePlaylistDialog = MutableStateFlow(false)
val showCreatePlaylistDialog: StateFlow<Boolean> = _showCreatePlaylistDialog.asStateFlow()

private val _pendingItemForPlaylist = MutableStateFlow<PendingPlaylistItemDetails?>(null)
```

```
private val _showSelectPlaylistDialog = MutableStateFlow(false)
val showSelectPlaylistDialog: StateFlow<Boolean> = _showSelectPlaylistDialog.asStateFlow()

fun openCreatePlaylistDialog() {
    _showCreatePlaylistDialog.value = true
}

fun closeCreatePlaylistDialog() {
    _showCreatePlaylistDialog.value = false
}

fun cancelAddToPlaylistFlow() {
    _showSelectPlaylistDialog.value = false
    _showCreatePlaylistDialog.value = false
    _pendingItemForPlaylist.value = null
}

fun prepareItemForPlaylistAddition(item: UnifiedDisplayItem) {
    _pendingItemForPlaylist.value = PendingPlaylistItemDetails(
        videoId = item.videoId,
        itemType = item.itemTypeForPlaylist,
        titleForDisplay = item.title,
        artistForDisplay = item.artistText,
        artworkForDisplay = item.artworkUrls.firstOrNull(),
        songStartSeconds = item.songStartSec,
        songEndSeconds = item.songEndSec,
        isExternal = item.isExternal // *** THE FIX: Get the flag from the item itself ***
    )
    _showSelectPlaylistDialog.value = true
    Timber.d("$TAG: Preparing item for playlist addition: ${_pendingItemForPlaylist.value}
}

fun addItemToExistingPlaylist(playlist: PlaylistEntity) {
    val pendingItem = _pendingItemForPlaylist.value ?: return cancelAddToPlaylistFlow()

    if (playlist.playlistId <= 0 && playlist.serverId == null) {
        Toast.makeText(application, "Cannot add items to this type of playlist.", Toast.LE
        return cancelAddToPlaylistFlow()
    }

    viewModelScope.launch {
        try {
            // Smart Dispatch: Only mark as DIRTY if the item is NOT external.
            if (!pendingItem.isExternal) {
                val updatedPlaylist = playlist.copy(
                    syncStatus = com.example.holodex.data.db.SyncStatus.DIRTY,
                    last_modified_at = Instant.now().toString()
                )
                holodexRepository.playlistDao.updatePlaylist(updatedPlaylist)
            }

            val lastOrder = holodexRepository.getLastItemOrderInPlaylist(playlist.playlist
            val newOrder = (lastOrder ?: -1) + 1
```

```
            val playlistItemEntity = PlaylistItemEntity(
                playlistOwnerId = playlist.playlistId,
                itemIdInPlaylist = if (pendingItem.itemType == LikedItemType.SONG_SEGMENT
                    LikedItemEntity.generateSongItemId(pendingItem.videoId, pendingItem.so
                } else {
                    LikedItemEntity.generateVideoItemId(pendingItem.videoId)
                },
                videoIdForItem = pendingItem.videoId,
                itemTypeInPlaylist = pendingItem.itemType,
                songStartSecondsPlaylist = pendingItem.songStartSeconds,
                songEndSecondsPlaylist = pendingItem.songEndSeconds,
                songNamePlaylist = if (pendingItem.itemType == LikedItemType.SONG_SEGMENT)
                songArtistTextPlaylist = pendingItem.artistForDisplay,
                songArtworkUrlPlaylist = pendingItem.artworkForDisplay,
                itemOrder = newOrder,
                isLocalOnly = pendingItem.isExternal // CRITICAL: Set the flag here
            )

            holodexRepository.addPlaylistItem(playlistItemEntity)
            Toast.makeText(application, "'${pendingItem.titleForDisplay}' added to ${playl
        } catch (e: Exception) {
            Timber.e(e, "Failed to add item to playlist ${playlist.name}")
            Toast.makeText(application, "Failed to add to playlist: ${e.localizedMessage}"
        } finally {
            cancelAddToPlaylistFlow()
        }
    }
}

fun handleCreateNewPlaylistFromSelectionDialog() {
    _showSelectPlaylistDialog.value = false
    _showCreatePlaylistDialog.value = true
}

fun confirmCreatePlaylist(name: String, description: String?) {
    val playlistName = name.trim()
    if (playlistName.isBlank()) {
        Toast.makeText(application, "Playlist name cannot be empty.", Toast.LENGTH_SHORT).
        return
    }

    val currentPendingItem = _pendingItemForPlaylist.value
    viewModelScope.launch {
        try {
            val newPlaylistId = holodexRepository.createNewPlaylist(playlistName, descript
            Toast.makeText(application, "Playlist '$playlistName' created", Toast.LENGTH_S
            _showCreatePlaylistDialog.value = false

            if (currentPendingItem != null) {
                addItemToExistingPlaylist(
                    PlaylistEntity(
                        playlistId = newPlaylistId, name = playlistName, description = des
                        createdAt = Instant.now().toString(), last_modified_at = Instant.n
                        serverId = null, owner = null
                    )
```

```
                    )
                }
            } catch (e: Exception) {
                Timber.e(e, "Failed to create playlist '$playlistName'")
                Toast.makeText(application, "Failed to create playlist: ${e.localizedMessage}"
            }
        }
    }

    fun deletePlaylist(playlist: PlaylistEntity) {
        val idToDelete = playlist.playlistId
        // This logic handles starred playlists (negative ID, has serverId) and user playlists
        if (idToDelete == 0L || (idToDelete < 0 && playlist.serverId == null)) {
            Toast.makeText(application, "Cannot delete synthetic playlists.", Toast.LENGTH_SHO
            return
        }
        viewModelScope.launch {
            try {
                holodexRepository.deletePlaylist(idToDelete)
                Toast.makeText(application, "Playlist deleted", Toast.LENGTH_SHORT).show()
            } catch (e: Exception) {
                Timber.e(e, "Failed to delete playlist ID: $idToDelete")
                Toast.makeText(application, "Failed to delete playlist: ${e.localizedMessage}"
            }
        }
    }
    fun prepareItemForPlaylistAdditionFromPlaybackItem(item: PlaybackItem) {
        _pendingItemForPlaylist.value = PendingPlaylistItemDetails(
            videoId = item.videoId,
            itemType = if (item.songId != null) LikedItemType.SONG_SEGMENT else LikedItemType.
            titleForDisplay = item.title,
            artistForDisplay = item.artistText,
            artworkForDisplay = item.artworkUri,
            songStartSeconds = item.clipStartSec?.toInt(),
            songEndSeconds = item.clipEndSec?.toInt(),
            isExternal = item.isExternal
        )
        _showSelectPlaylistDialog.value = true
    }
}

// File: java\com\example\holodex\viewmodel\SettingsViewModel.kt
package com.example.holodex.viewmodel

import android.app.Application
import android.content.Intent
import android.content.SharedPreferences
import android.net.Uri
import androidx.core.content.edit
import androidx.core.net.toUri
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import androidx.media3.common.util.UnstableApi
import androidx.work.Constraints
import androidx.work.ExistingWorkPolicy
```

```
import androidx.work.NetworkType
import androidx.work.OneTimeWorkRequestBuilder
import androidx.work.WorkManager
import com.example.holodex.MyApp
import com.example.holodex.background.SyncWorker
import com.example.holodex.data.download.LegacyDownloadScanner
import com.example.holodex.data.repository.UserPreferencesRepository
import dagger.hilt.android.lifecycle.HiltViewModel
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.SharingStarted
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.flow.stateIn
import kotlinx.coroutines.launch
import timber.log.Timber
import javax.inject.Inject

sealed class ApiKeySaveResult {
    object Success : ApiKeySaveResult()
    object Empty : ApiKeySaveResult()
    data class Error(val message: String) : ApiKeySaveResult()
    object Idle : ApiKeySaveResult()
}

object ThemePreference {
    const val KEY = "app_theme_preference"
    const val LIGHT = "LIGHT"
    const val DARK = "DARK"
    const val SYSTEM = "SYSTEM"
}

// Centralized preference constants
object AppPreferenceConstants {
    // Image Quality
    const val PREF_IMAGE_QUALITY = "pref_image_quality"
    const val IMAGE_QUALITY_AUTO = "AUTO"
    const val IMAGE_QUALITY_MEDIUM = "MEDIUM"
    const val IMAGE_QUALITY_LOW = "LOW"

    // Audio Quality (for NewPipeExtractor)
    const val PREF_AUDIO_QUALITY = "pref_audio_quality"
    const val AUDIO_QUALITY_BEST = "BEST"
    const val AUDIO_QUALITY_STANDARD = "STANDARD"
    const val AUDIO_QUALITY_SAVER = "SAVER"

    // Paging Configuration (Data Loading Intensity for Lists)
    const val PREF_LIST_LOADING_CONFIG = "pref_list_loading_config"
    const val LIST_LOADING_NORMAL = "NORMAL"
    const val LIST_LOADING_REDUCED = "REDUCED"
    const val LIST_LOADING_MINIMAL = "MINIMAL"

    // ExoPlayer Buffering Strategy
    const val PREF_BUFFERING_STRATEGY = "pref_buffering_strategy"
    const val BUFFERING_STRATEGY_AGGRESSIVE = "AGGRESSIVE_START"
    const val BUFFERING_STRATEGY_BALANCED = "BALANCED"
```

```
    const val BUFFERING_STRATEGY_STABLE = "STABLE_PLAYBACK"

    const val PREF_AUTOPLAY_NEXT_VIDEO = "pref_autoplay_next_video"
    const val PREF_DOWNLOAD_LOCATION = "pref_download_location_uri"

}
sealed class ScanStatus {
    object Idle : ScanStatus()
    object Scanning : ScanStatus()
    data class Complete(val importedCount: Int) : ScanStatus()
    data class Error(val message: String) : ScanStatus()
}

@HiltViewModel
class SettingsViewModel @Inject constructor(
    private val application: Application,
    private val sharedPreferences: SharedPreferences,
    private val userPreferencesRepository: UserPreferencesRepository,
    private val workManager: WorkManager,
    private val legacyDownloadScanner: LegacyDownloadScanner

) : ViewModel() {

    private val _apiKeySaveResult = MutableStateFlow<ApiKeySaveResult>(ApiKeySaveResult.Idle)
    val apiKeySaveResult: StateFlow<ApiKeySaveResult> = _apiKeySaveResult.asStateFlow()

    private val _currentApiKey = MutableStateFlow<String>(loadInitialApiKey())
    val currentApiKey: StateFlow<String> = _currentApiKey.asStateFlow()

    private val _cacheClearStatus = MutableStateFlow<String?>(null)
    val cacheClearStatus: StateFlow<String?> = _cacheClearStatus.asStateFlow()

    private val _currentThemePreference = MutableStateFlow(loadThemePreference())
    val currentThemePreference: StateFlow<String> = _currentThemePreference.asStateFlow()

    private val _currentImageQuality = MutableStateFlow(loadImageQualityPreference())
    val currentImageQuality: StateFlow<String> = _currentImageQuality.asStateFlow()

    private val _currentAudioQuality = MutableStateFlow(loadAudioQualityPreference())
    val currentAudioQuality: StateFlow<String> = _currentAudioQuality.asStateFlow()

    private val _currentListLoadingConfig = MutableStateFlow(loadListLoadingConfigPreference()
    val currentListLoadingConfig: StateFlow<String> = _currentListLoadingConfig.asStateFlow()

    private val _currentBufferingStrategy = MutableStateFlow(loadBufferingStrategyPreference()
    val currentBufferingStrategy: StateFlow<String> = _currentBufferingStrategy.asStateFlow()

    private val _downloadLocation = MutableStateFlow(loadDownloadLocation())
    val downloadLocation: StateFlow<String> = _downloadLocation.asStateFlow()

    private val _scanStatus = MutableStateFlow<ScanStatus>(ScanStatus.Idle)
    val scanStatus: StateFlow<ScanStatus> = _scanStatus.asStateFlow()

    val autoplayNextVideoEnabled: StateFlow<Boolean> = userPreferencesRepository.autoplayEnabl
        .stateIn(
```

```
        scope = viewModelScope,
        started = SharingStarted.WhileSubscribed(5000L),
        initialValue = true // Default value should match what's in UserPreferencesReposit
    )
val shuffleOnPlayStartEnabled: StateFlow<Boolean> = userPreferencesRepository.shuffleOnPla
    .stateIn(
        scope = viewModelScope,
        started = SharingStarted.WhileSubscribed(5000L),
        initialValue = false // Matches the repository default
    )

init {
    Timber.d("SettingsViewModel initialized.")
    // The collectLatest below is now redundant for the autoplayNextVideoEnabled flow itse
    // as stateIn already makes it hot and provides the latest value.
    // It could be kept if you needed to perform side-effects *every time* the preference
    // within the ViewModel's scope, but usually, the UI collects it directly.
    /*
    viewModelScope.launch {
        userPreferencesRepository.autoplayEnabled.collectLatest { enabled ->
            Timber.d("SettingsViewModel: Autoplay preference updated to $enabled.")
        }
    }
    */
}
fun runLegacyFileScan() {
    if (_scanStatus.value is ScanStatus.Scanning) return // Prevent multiple scans

    viewModelScope.launch {
        _scanStatus.value = ScanStatus.Scanning
        try {
            val count = legacyDownloadScanner.scanAndImportLegacyDownloads()
            _scanStatus.value = ScanStatus.Complete(count)
        } catch (e: Exception) {
            Timber.e(e, "Legacy scan failed in ViewModel")
            _scanStatus.value = ScanStatus.Error("Scan failed: ${e.localizedMessage}")
        }
    }
}

fun resetScanStatus() {
    _scanStatus.value = ScanStatus.Idle
}
fun triggerManualSync() {
    viewModelScope.launch {
        Timber.i("SettingsViewModel: Manual sync triggered by user.")
        _transientMessage.emit("Syncing account data...") // Use the existing message flow

        val constraints = Constraints.Builder()
            .setRequiredNetworkType(NetworkType.CONNECTED)
            .build()

        val immediateSyncRequest = OneTimeWorkRequestBuilder<SyncWorker>()
            .setConstraints(constraints)
            .build()
```

```
        // Enqueue as unique work with REPLACE to ensure it runs now,
        // even if a periodic one was scheduled soon.
        workManager.enqueueUniqueWork(
            "ManualSync",
            ExistingWorkPolicy.REPLACE,
            immediateSyncRequest
        )
    }
}
private val _transientMessage = MutableStateFlow<String?>(null)
val transientMessage: StateFlow<String?> = _transientMessage.asStateFlow()

fun clearTransientMessage() {
    _transientMessage.value = null
}
private fun loadDownloadLocation(): String {
    return sharedPreferences.getString(AppPreferenceConstants.PREF_DOWNLOAD_LOCATION, "")
}

fun saveDownloadLocation(uri: Uri) {
    viewModelScope.launch {
        try {
            // Take persistent permission to access the folder
            val contentResolver = application.contentResolver
            val flags = Intent.FLAG_GRANT_READ_URI_PERMISSION or Intent.FLAG_GRANT_WRITE_U
            contentResolver.takePersistableUriPermission(uri, flags)

            val uriString = uri.toString()
            sharedPreferences.edit { putString(AppPreferenceConstants.PREF_DOWNLOAD_LOCATI
            _downloadLocation.value = uriString
            Timber.i("Saved new download location URI: $uriString")
        } catch (e: Exception) {
            Timber.e(e, "Failed to save download location permission or preference.")
            // Optionally emit an error to a snackbar/toast
        }
    }
}

fun clearDownloadLocation() {
    viewModelScope.launch {
        val currentUriString = _downloadLocation.value
        if (currentUriString.isNotEmpty()) {
            try {
                val uri = currentUriString.toUri()
                val contentResolver = application.contentResolver
                val flags = Intent.FLAG_GRANT_READ_URI_PERMISSION or Intent.FLAG_GRANT_WRI
                contentResolver.releasePersistableUriPermission(uri, flags)
                Timber.i("Released persistable URI permission for: $currentUriString")
            } catch (e: Exception) {
                Timber.e(e, "Failed to release persistable URI permission.")
            }
        }
        sharedPreferences.edit { remove(AppPreferenceConstants.PREF_DOWNLOAD_LOCATION) }
        _downloadLocation.value = ""
```

```
    }
}

private fun loadInitialApiKey(): String {
    return sharedPreferences.getString("API_KEY", "") ?: ""
}

fun saveApiKey(key: String) {
    viewModelScope.launch {
        val trimmedKey = key.trim()
        if (trimmedKey.isBlank()) {
            _apiKeySaveResult.value = ApiKeySaveResult.Empty
            return@launch
        }
        try {
            sharedPreferences.edit { putString("API_KEY", trimmedKey) }
            _currentApiKey.value = trimmedKey
            _apiKeySaveResult.value = ApiKeySaveResult.Success
        } catch (e: Exception) {
            Timber.tag("SettingsViewModel").e(e, "Error saving API key")
            _apiKeySaveResult.value = ApiKeySaveResult.Error("Failed to save API key.")
        }
    }
}

fun resetApiKeySaveResult() {
    _apiKeySaveResult.value = ApiKeySaveResult.Idle
}

@UnstableApi
fun clearAllApplicationCaches() {
    _cacheClearStatus.value = "Clearing caches..."
    (application as? MyApp)?.clearAllAppCachesOnDemand { success ->
        viewModelScope.launch {
            if (success) {
                _cacheClearStatus.value = "All caches cleared successfully."
                Timber.i("All caches cleared successfully reported by MyApp.")
            } else {
                _cacheClearStatus.value = "Failed to clear all caches."
                Timber.e("Cache clearing failed as reported by MyApp.")
            }
        }
    } ?: run {
        viewModelScope.launch {
            _cacheClearStatus.value = "Error: Could not access application to clear caches
            Timber.e("Application context is not MyApp instance or is null.")
        }
    }
}

fun resetCacheClearStatus() {
    _cacheClearStatus.value = null
}

private fun loadThemePreference(): String {
```

```
    return sharedPreferences.getString(ThemePreference.KEY, ThemePreference.SYSTEM)
        ?: ThemePreference.SYSTEM
}

fun setThemePreference(themeValue: String) {
    if (themeValue !in listOf(
            ThemePreference.LIGHT,
            ThemePreference.DARK,
            ThemePreference.SYSTEM
        )
    ) {
        Timber.w("Invalid theme value set: $themeValue. Defaulting to SYSTEM.")
        _currentThemePreference.value = ThemePreference.SYSTEM
        sharedPreferences.edit { putString(ThemePreference.KEY, ThemePreference.SYSTEM) }
        return
    }
    viewModelScope.launch {
        _currentThemePreference.value = themeValue
        sharedPreferences.edit {
            putString(ThemePreference.KEY, themeValue)
        }
        Timber.d("Theme preference saved: $themeValue")
    }
}

private fun loadImageQualityPreference(): String {
    return sharedPreferences.getString(
        AppPreferenceConstants.PREF_IMAGE_QUALITY,
        AppPreferenceConstants.IMAGE_QUALITY_AUTO
    ) ?: AppPreferenceConstants.IMAGE_QUALITY_AUTO
}

fun setImageQualityPreference(quality: String) {
    viewModelScope.launch {
        _currentImageQuality.value = quality
        sharedPreferences.edit { putString(AppPreferenceConstants.PREF_IMAGE_QUALITY, qual
        Timber.d("Image quality preference saved: $quality. App restart may be needed for
    }
}

private fun loadAudioQualityPreference(): String {
    return sharedPreferences.getString(
        AppPreferenceConstants.PREF_AUDIO_QUALITY,
        AppPreferenceConstants.AUDIO_QUALITY_BEST
    ) ?: AppPreferenceConstants.AUDIO_QUALITY_BEST
}

fun setAudioQualityPreference(quality: String) {
    viewModelScope.launch {
        _currentAudioQuality.value = quality
        sharedPreferences.edit { putString(AppPreferenceConstants.PREF_AUDIO_QUALITY, qual
        Timber.d("Audio quality preference saved: $quality")
    }
}
```

```
    // NEW: Function to set autoplay preference
    fun setAutoplayNextVideoEnabled(enabled: Boolean) {
        viewModelScope.launch {
            userPreferencesRepository.setAutoplayEnabled(enabled)
            Timber.d("Autoplay next video preference set to: $enabled")
        }
    }
    fun setShuffleOnPlayStartEnabled(enabled: Boolean) {
        viewModelScope.launch {
            userPreferencesRepository.setShuffleOnPlayStartEnabled(enabled)
        }
    }
    private fun loadListLoadingConfigPreference(): String {
        return sharedPreferences.getString(
            AppPreferenceConstants.PREF_LIST_LOADING_CONFIG,
            AppPreferenceConstants.LIST_LOADING_NORMAL
        ) ?: AppPreferenceConstants.LIST_LOADING_NORMAL
    }

    fun setListLoadingConfigPreference(config: String) {
        viewModelScope.launch {
            _currentListLoadingConfig.value = config
            sharedPreferences.edit {
                putString(
                    AppPreferenceConstants.PREF_LIST_LOADING_CONFIG,
                    config
                )
            }
            Timber.d("List loading config saved: $config. HolodexRepository will need to react
        }
    }

    private fun loadBufferingStrategyPreference(): String {
        return sharedPreferences.getString(
            AppPreferenceConstants.PREF_BUFFERING_STRATEGY,
            AppPreferenceConstants.BUFFERING_STRATEGY_AGGRESSIVE
        ) ?: AppPreferenceConstants.BUFFERING_STRATEGY_AGGRESSIVE
    }

    fun setBufferingStrategyPreference(strategy: String) {
        viewModelScope.launch {
            _currentBufferingStrategy.value = strategy
            sharedPreferences.edit {
                putString(
                    AppPreferenceConstants.PREF_BUFFERING_STRATEGY,
                    strategy
                )
            }
            Timber.d("Buffering strategy saved: $strategy. ExoPlayer will need to be re-create
        }
    }
}

// File: java\com\example\holodex\viewmodel\SharedViewModelTypes.kt
package com.example.holodex.viewmodel
```

```
import androidx.compose.runtime.MutableState
import androidx.compose.runtime.mutableStateOf
import kotlinx.coroutines.Job

data class SubOrgHeader(val name: String)

enum class MusicCategoryType {
    LATEST,
    UPCOMING_MUSIC,
    SEARCH,
    FAVORITES,
    LIKED_SEGMENTS,
    TRENDING,
    RECENT_STREAMS,
    COMMUNITY_PLAYLISTS,
    ARTIST_RADIOS,
    SYSTEM_PLAYLISTS,
    DISCOVER_CHANNELS
}

// Helper for manual pagination (used by FullListViewModel)
class ListStateHolder<T> {
    val items: MutableState<List<T>> = mutableStateOf(emptyList())
    val isLoadingInitial: MutableState<Boolean> = mutableStateOf(false)
    val isLoadingMore: MutableState<Boolean> = mutableStateOf(false)
    val endOfList: MutableState<Boolean> = mutableStateOf(false)
    var currentOffset: Int = 0
    var job: Job? = null
}

// File: java\com\example\holodex\viewmodel\UnifiedDisplayItem.kt
package com.example.holodex.viewmodel

import androidx.compose.runtime.Immutable
import com.example.holodex.data.db.LikedItemType

/**
 * A single, canonical data class for any item displayed in a list.
 * It contains all possible fields the UI might need, abstracting away the
 * original data source (video, liked item, history, download, etc.).
 * Marked as Immutable for Compose performance optimization.
 */
@Immutable
data class UnifiedDisplayItem(
    // Core Identifiers
    val stableId: String, // A unique ID for this item in a list (e.g., "history_12345", "like
    val playbackItemId: String, // The ID used for playback and liking (e.g., "videoId_start"
    val videoId: String,
    val channelId: String,

    // Display Fields
    val title: String,
    val artistText: String,
    val artworkUrls: List<String>, // A prioritized list of URLs for Coil to try
```

```
    val durationText: String,

    // Metadata & Badges
    val songCount: Int?, // For full videos, null for segments
    val isDownloaded: Boolean,
    val isSegment: Boolean,
    val isLiked: Boolean,

    // Data for Actions
    val itemTypeForPlaylist: LikedItemType, // Is it a VIDEO or a SONG_SEGMENT?
    val songStartSec: Int?,
    val songEndSec: Int?,
    val originalArtist: String?,
    val isExternal: Boolean
)

// File: java\com\example\holodex\viewmodel\VideoDetailsViewModel.kt
// File: java/com/example/holodex/viewmodel/VideoDetailsViewModel.kt
package com.example.holodex.viewmodel

import androidx.compose.ui.graphics.Color
import androidx.lifecycle.SavedStateHandle
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import androidx.media3.common.util.UnstableApi
import com.example.holodex.data.download.NoDownloadLocationException
import com.example.holodex.data.model.HolodexSong
import com.example.holodex.data.model.HolodexVideoItem
import com.example.holodex.data.repository.DownloadRepository
import com.example.holodex.data.repository.HolodexRepository
import com.example.holodex.playback.PlaybackRequestManager
import com.example.holodex.playback.domain.model.PlaybackItem
import com.example.holodex.playback.domain.usecase.AddOrFetchAndAddUseCase
import com.example.holodex.util.DynamicTheme
import com.example.holodex.util.PaletteExtractor
import com.example.holodex.util.ThumbnailQuality
import com.example.holodex.util.getYouTubeThumbnailUrl
import com.example.holodex.viewmodel.mappers.toPlaybackItem
import com.example.holodex.viewmodel.mappers.toUnifiedDisplayItem
import com.example.holodex.viewmodel.mappers.toVirtualSegmentUnifiedDisplayItem
import dagger.hilt.android.lifecycle.HiltViewModel
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.flow.first
import kotlinx.coroutines.launch
import timber.log.Timber
import javax.inject.Inject

@UnstableApi
@HiltViewModel
class VideoDetailsViewModel @Inject constructor(
    private val savedStateHandle: SavedStateHandle,
    private val holodexRepository: HolodexRepository,
    private val downloadRepository: DownloadRepository,
```

```
    private val playbackRequestManager: PlaybackRequestManager,
    private val addOrFetchAndAddUseCase: AddOrFetchAndAddUseCase,
    private val paletteExtractor: PaletteExtractor
) : ViewModel() {

    companion object {
        const val VIDEO_ID_ARG = "videoId"
        private const val TAG = "VideoDetailsVM"
    }

    val videoId: String = savedStateHandle.get<String>(VIDEO_ID_ARG) ?: ""

    private val _videoDetails = MutableStateFlow<HolodexVideoItem?>(null)
    val videoDetails: StateFlow<HolodexVideoItem?> = _videoDetails.asStateFlow()

    private val _isLoading = MutableStateFlow(true)
    val isLoading: StateFlow<Boolean> = _isLoading.asStateFlow()

    private val _error = MutableStateFlow<String?>(null)
    val error: StateFlow<String?> = _error.asStateFlow()

    private val _transientMessage = MutableStateFlow<String?>(null)
    val transientMessage: StateFlow<String?> = _transientMessage.asStateFlow()

    private val _unifiedSongItems = MutableStateFlow<List<UnifiedDisplayItem>>(emptyList())
    val unifiedSongItems: StateFlow<List<UnifiedDisplayItem>> = _unifiedSongItems.asStateFlow(

    private val _dynamicTheme = MutableStateFlow(DynamicTheme.default(Color.Black, Color.White
    val dynamicTheme: StateFlow<DynamicTheme> = _dynamicTheme.asStateFlow()

    fun initialize(videoListViewModel: VideoListViewModel) {
        if (videoId.isNotBlank()) {
            viewModelScope.launch {
                _isLoading.value = true
                val prefetchedItem = videoListViewModel.videoItemForDetailScreen.value

                val isPrefetchedItemComplete = prefetchedItem != null &&
                        prefetchedItem.id == videoId &&
                        (prefetchedItem.channel.org == "External" || prefetchedItem.songs != n

                if (isPrefetchedItemComplete) {
                    Timber.d("Using COMPLETE pre-fetched video data for $videoId")
                    processItem(prefetchedItem!!) // We know it's not null here
                } else {
                    Timber.d("Pre-fetched data for $videoId is INCOMPLETE or missing. Fetching
                    holodexRepository.getVideoWithSongs(videoId, forceRefresh = false)
                        .onSuccess { processItem(it) }
                        .onFailure { _error.value = "Failed to load video details: ${it.locali
                }
                _isLoading.value = false
            }
        } else {
            _isLoading.value = false
            _error.value = "Video ID is missing."
        }
```

```
}

// A single, unified processing function
private fun processItem(videoItem: HolodexVideoItem) {
    _videoDetails.value = videoItem
    viewModelScope.launch {
        updateTheme(videoItem.id)

        // *** THE CORRECTED LOGIC ***
        // Check if the item is explicitly marked as External OR if it's a Holodex item wi
        val isEffectivelySegmentless = videoItem.channel.org == "External" || videoItem.so

        if (isEffectivelySegmentless) {
            // Process as a single virtual segment
            val likedIds = holodexRepository.likedItemIds.first()
            val isLiked = likedIds.contains(videoItem.id)
            val virtualSegment = videoItem.toVirtualSegmentUnifiedDisplayItem(isLiked, isD
            _unifiedSongItems.value = listOf(virtualSegment)
        } else {
            // Process the existing song list from the Holodex item
            val songs = videoItem.songs!! // We know it's not null or empty here
            val likedIds = holodexRepository.likedItemIds.first()
            val downloadedIds = downloadRepository.getAllDownloads().first().map { it.vide
            val unifiedItems = songs.sortedBy { it.start }.map { song ->
                song.toUnifiedDisplayItem(
                    parentVideo = videoItem,
                    isLiked = likedIds.contains("${videoItem.id}_${song.start}"),
                    isDownloaded = downloadedIds.contains("${videoItem.id}_${song.start}")
                )
            }
            _unifiedSongItems.value = unifiedItems
        }
    }
}

private suspend fun updateTheme(videoId: String) {
    val artworkUrl = getYouTubeThumbnailUrl(videoId, ThumbnailQuality.MAX).firstOrNull()
    _dynamicTheme.value = paletteExtractor.extractThemeFromUrl(
        artworkUrl,
        DynamicTheme.default(Color.Black, Color.White)
    )
}

fun playAllSegments() { playSegment(0) }

fun playSegment(startIndex: Int) {
    viewModelScope.launch {
        // This now works for both cases, as _unifiedSongItems is always populated correct
        val itemsToPlay = _unifiedSongItems.value.map { it.toPlaybackItem() }
        if (startIndex in itemsToPlay.indices) {
            playbackRequestManager.submitPlaybackRequest(items = itemsToPlay, startIndex =
        } else {
            _error.value = "Invalid song index."
        }
    }
```

```
}

fun addAllSegmentsToQueue() {
    viewModelScope.launch {
        val itemsToAdd = _unifiedSongItems.value
        if (itemsToAdd.isNotEmpty()) {
            addOrFetchAndAddUseCase(itemsToAdd.first().toPlaybackItem().copy(songId = null
                .onSuccess { message -> _transientMessage.value = message }
                .onFailure { error -> _error.value = "Failed to add to queue: ${error.loca
        }
    }
}

fun downloadAllSegments() {
    val video = _videoDetails.value
    if (video == null || video.songs.isNullOrEmpty()) {
        _error.value = "No segments available to download."
        return
    }
    viewModelScope.launch {
        try {
            _transientMessage.value = "Queueing ${video.songs.size} songs for download..."
            video.songs.forEach { song ->
                downloadRepository.startDownload(video, song)
            }
        } catch (e: Exception) {
            Timber.e(e, "A general error occurred during bulk download initiation.")
            _error.value = "Could not start downloads: An unknown error occurred."
        }
    }
}

fun requestDownloadForSongFromPlaybackItem(item: PlaybackItem) {
    viewModelScope.launch {
        val videoResult = holodexRepository.getVideoWithSongs(item.videoId, false).getOrNu
        if (videoResult == null) {
            _error.value = "Could not find video details to start download."
            return@launch
        }
        val songToDownload = videoResult.songs?.find { it.start.toLong() == item.clipStart
        if (songToDownload == null) {
            _error.value = "Could not find matching song segment to download."
            return@launch
        }
        requestDownloadForSong(videoResult, songToDownload)
    }
}

fun requestDownloadForSong(videoItem: HolodexVideoItem, song: HolodexSong) {
    viewModelScope.launch {
        try {
            downloadRepository.startDownload(videoItem, song)
            _transientMessage.value = "Added '${song.name}' to download queue."
        } catch (e: NoDownloadLocationException) {
            _error.value = e.message
```

```
            } catch (e: Exception) {
                _error.value = "Could not start download: An unknown error occurred."
            }
        }
    }

    fun clearError() {
        _error.value = null
    }

    fun clearTransientMessage() {
        _transientMessage.value = null
    }
}

// File: java\com\example\holodex\viewmodel\VideoListViewModel.kt
// File: java/com/example/holodex/viewmodel/VideoListViewModel.kt
package com.example.holodex.viewmodel

import android.content.SharedPreferences
import androidx.core.content.edit
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import androidx.media3.common.util.UnstableApi
import com.example.holodex.data.cache.BrowseCacheKey
import com.example.holodex.data.cache.FetcherResult
import com.example.holodex.data.cache.SearchCacheKey
import com.example.holodex.data.model.HolodexSong
import com.example.holodex.data.model.HolodexVideoItem
import com.example.holodex.data.repository.DownloadRepository
import com.example.holodex.data.repository.HolodexRepository
import com.example.holodex.data.repository.HolodexRepository.Companion.DEFAULT_PAGE_SIZE
import com.example.holodex.data.repository.LocalRepository
import com.example.holodex.data.repository.SearchHistoryRepository
import com.example.holodex.playback.PlaybackRequestManager
import com.example.holodex.playback.domain.model.PlaybackItem
import com.example.holodex.playback.domain.usecase.AddOrFetchAndAddUseCase
import com.example.holodex.util.extractVideoIdFromQuery
import com.example.holodex.viewmodel.autoplay.AutoplayItemProvider
import com.example.holodex.viewmodel.autoplay.ContinuationManager
import com.example.holodex.viewmodel.mappers.toUnifiedDisplayItem
import com.example.holodex.viewmodel.state.BrowseFilterState
import com.example.holodex.viewmodel.state.SongSegmentFilterMode
import com.example.holodex.viewmodel.state.ViewTypePreset
import com.google.gson.Gson
import dagger.hilt.android.lifecycle.HiltViewModel
import kotlinx.coroutines.FlowPreview
import kotlinx.coroutines.Job
import kotlinx.coroutines.async
import kotlinx.coroutines.flow.MutableSharedFlow
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.SharedFlow
import kotlinx.coroutines.flow.SharingStarted
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asSharedFlow
```

```
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.flow.collectLatest
import kotlinx.coroutines.flow.combine
import kotlinx.coroutines.flow.debounce
import kotlinx.coroutines.flow.distinctUntilChanged
import kotlinx.coroutines.flow.first
import kotlinx.coroutines.flow.map
import kotlinx.coroutines.flow.stateIn
import kotlinx.coroutines.launch
import kotlinx.coroutines.sync.Mutex
import kotlinx.coroutines.sync.withLock
import org.orbitmvi.orbit.ContainerHost
import org.orbitmvi.orbit.syntax.simple.intent
import org.orbitmvi.orbit.syntax.simple.postSideEffect
import org.orbitmvi.orbit.syntax.simple.reduce
import org.orbitmvi.orbit.viewmodel.container
import timber.log.Timber
import javax.inject.Inject

enum class MusicCategoryType {
    LATEST,
    UPCOMING_MUSIC,
    SEARCH,
    FAVORITES,
    LIKED_SEGMENTS,
    TRENDING,
    RECENT_STREAMS,
    COMMUNITY_PLAYLISTS,
    ARTIST_RADIOS,
    SYSTEM_PLAYLISTS,
    DISCOVER_CHANNELS
}

// 1. Define the state
data class VideoListState(
    val isInitialized: Boolean = false,
    val videoItemForDetailScreen: HolodexVideoItem? = null,
    val selectedOrganization: String = "Nijisanji",
    val browseList: List<UnifiedDisplayItem> = emptyList(),
    val searchList: List<UnifiedDisplayItem> = emptyList(),
    val browseIsLoadingInitial: Boolean = false,
    val browseIsLoadingMore: Boolean = false,
    val browseIsRefreshing: Boolean = false,
    val browseEndOfList: Boolean = false,
    val searchIsLoadingInitial: Boolean = false,
    val searchIsLoadingMore: Boolean = false,
    val searchEndOfList: Boolean = false,
    val errorMessage: String? = null,
    val isLoadingGeneral: Boolean = false,
    val isSearchActive: Boolean = false,
    val browseFilterState: BrowseFilterState = BrowseFilterState.create(
        ViewTypePreset.UPCOMING_STREAMS,
        SongSegmentFilterMode.ALL,
    ),
    val currentSearchQuery: String = "",
```

```
    val activeListContextType: MusicCategoryType = MusicCategoryType.LATEST,
    val searchHistory: List<String> = emptyList(),
    val availableOrganizations: List<Pair<String, String?>> = emptyList(),
    val activeSearchSource: String = "Holodex",
)

// 2. Define the side effects
sealed class VideoListSideEffect {
    data class ShowToast(val message: String) : VideoListSideEffect()
    data class Navigation(val destination: NavigationDestination) : VideoListSideEffect()
}

sealed class NavigationDestination {
    data class VideoDetails(val videoId: String) : NavigationDestination()
    object HomeScreenWithSearch : NavigationDestination()
}


enum class MusicCategoryType {
    LATEST,
    UPCOMING_MUSIC,
    SEARCH,
    FAVORITES,
    LIKED_SEGMENTS,
    TRENDING,
    RECENT_STREAMS,
    COMMUNITY_PLAYLISTS,
    ARTIST_RADIOS,
    SYSTEM_PLAYLISTS,
    DISCOVER_CHANNELS
}

@UnstableApi
@OptIn(FlowPreview::class)
@HiltViewModel
class VideoListViewModel @androidx.annotation.OptIn(UnstableApi::class)
@Inject constructor(
    private val holodexRepository: HolodexRepository,
    private val localRepository: LocalRepository,
    private val sharedPreferences: SharedPreferences,
    private val searchHistoryRepository: SearchHistoryRepository,
    private val playbackRequestManager: PlaybackRequestManager,
    private val downloadRepository: DownloadRepository,
    private val continuationManager: ContinuationManager,
    private val addOrFetchAndAddUseCase: AddOrFetchAndAddUseCase
) : ViewModel(), ContainerHost<VideoListState, VideoListSideEffect> {

    override val container = container<VideoListState, VideoListSideEffect>(
        initialState = VideoListState(),
        buildSettings = {
            exceptionHandler = {
                Timber.e(it)
                intent {
                    postSideEffect(VideoListSideEffect.ShowToast("Error: ${it.message}"))
                }
```

```
            null
        }
    }
)

private val isInitialized = MutableStateFlow(false)
private val _transientMessage = MutableSharedFlow<String>()
val transientMessage: SharedFlow<String> = _transientMessage.asSharedFlow()

private val _videoItemForDetailScreen = MutableStateFlow<HolodexVideoItem?>(null)
val videoItemForDetailScreen: StateFlow<HolodexVideoItem?> = _videoItemForDetailScreen.asS

private val _selectedOrganization = MutableStateFlow(
    sharedPreferences.getString(PREF_LAST_SELECTED_ORG, "Nijisanji") ?: "Hololive"
)
val selectedOrganization: StateFlow<String> = _selectedOrganization.asStateFlow()

// --- STEP 1: Define holders for RAW data, not the final UI data ---
private val rawBrowseListState = ListStateHolder<HolodexVideoItem>()
private val rawSearchListState = ListStateHolder<HolodexVideoItem>()

// --- STEP 2: Create the NEW REACTIVE StateFlows for the UI ---
val browseListState: StateFlow<List<UnifiedDisplayItem>> = combine(
    rawBrowseListState.items,
    holodexRepository.likedItemIds,
    downloadRepository.getAllDownloads().map { list -> list.map { it.videoId }.toSet() }
) { rawItems, likedIds, downloadedIds ->
    rawItems.map { video ->
        video.toUnifiedDisplayItem(
            isLiked = likedIds.contains(video.id),
            downloadedSegmentIds = downloadedIds
        )
    }
}.stateIn(viewModelScope, SharingStarted.WhileSubscribed(5000L), emptyList())

val searchListState: StateFlow<List<UnifiedDisplayItem>> = combine(
    rawSearchListState.items,
    holodexRepository.likedItemIds,
    downloadRepository.getAllDownloads().map { list -> list.map { it.videoId }.toSet() }
) { rawItems, likedIds, downloadedIds ->
    rawItems.map { video ->
        video.toUnifiedDisplayItem(
            isLiked = likedIds.contains(video.id),
            downloadedSegmentIds = downloadedIds
        )
    }
}.stateIn(viewModelScope, SharingStarted.WhileSubscribed(5000L), emptyList())

// --- We also need StateFlows for the loading/pagination states for the UI ---
val browseIsLoadingInitial = rawBrowseListState.isLoadingInitial
val browseIsLoadingMore = rawBrowseListState.isLoadingMore
val browseIsRefreshing = rawBrowseListState.isRefreshingViaPull
val browseEndOfList = rawBrowseListState.endOfList

val searchIsLoadingInitial = rawSearchListState.isLoadingInitial
```

```
val searchIsLoadingMore = rawSearchListState.isLoadingMore
val searchEndOfList = rawSearchListState.endOfList
fun setOrganization(orgName: String) {
    if (_selectedOrganization.value != orgName) {
        _selectedOrganization.value = orgName
        // Save the new selection to SharedPreferences
        sharedPreferences.edit {
            putString(PREF_LAST_SELECTED_ORG, orgName)
        }
    }
}

companion object {
    const val TAG = "VideoListViewModel"
    const val PREF_LAST_SELECTED_ORG = "last_selected_org"
    const val PREF_LAST_CATEGORY_TYPE = "last_category_type"
    const val PREF_LAST_SEARCH_QUERY = "last_search_query"
    const val PREF_LAST_BROWSE_FILTERS = "last_browse_filters_v1"
    const val CHANNEL_ID_SEARCH_PREFIX = "channel:"
}

private val autoplayItemProvider: AutoplayItemProvider = AutoplayItemProvider(holodexRepos
val availableOrganizations: StateFlow<List<Pair<String, String?>>> =
    holodexRepository.availableOrganizations
private val _errorMessage = MutableStateFlow<String?>(null)
val errorMessage: StateFlow<String?> = _errorMessage.asStateFlow()
private val _isLoadingGeneral = MutableStateFlow(false)

data class ListStateHolder<T>(
    val items: MutableStateFlow<List<T>> = MutableStateFlow(emptyList()),
    val isLoadingInitial: MutableStateFlow<Boolean> = MutableStateFlow(false),
    val isLoadingMore: MutableStateFlow<Boolean> = MutableStateFlow(false),
    val isRefreshingViaPull: MutableStateFlow<Boolean> = MutableStateFlow(false),
    val endOfList: MutableStateFlow<Boolean> = MutableStateFlow(false),
    var currentOffset: Int = 0,
    var job: Job? = null,
    var currentKeyForPrefetch: Any? = null
)

sealed class NavigationDestination {
    data class VideoDetails(val videoId: String) : NavigationDestination()
    object HomeScreenWithSearch : NavigationDestination()
}

private val _navigationRequest = MutableStateFlow<NavigationDestination?>(null)
val navigationRequest: StateFlow<NavigationDestination?> = _navigationRequest.asStateFlow(

private val _isSearchActive = MutableStateFlow(false)
val isSearchActive: StateFlow<Boolean> = _isSearchActive.asStateFlow()

private val _browseFilterState = MutableStateFlow(loadLastBrowseFilters())
val browseFilterState: StateFlow<BrowseFilterState> = _browseFilterState.asStateFlow()

private val _currentSearchQuery = MutableStateFlow(loadLastSearchQuery())
val currentSearchQuery: StateFlow<String> = _currentSearchQuery.asStateFlow()
```

```
private val _activeListContextType = MutableStateFlow(loadLastActiveListContextType())
val activeListContextType: StateFlow<MusicCategoryType> = _activeListContextType.asStateFl

private val browseMutex = Mutex()
private val searchMutex = Mutex()


private fun showAutoplayNoMoreItemsMessage(context: MusicCategoryType) {
    _errorMessage.value = "Autoplay: No more items in the current '$context' list on scree
}

val searchHistory: StateFlow<List<String>> = searchHistoryRepository.searchHistory

val browseScreenCategories: List<Pair<String, BrowseFilterState>>
    get() = listOf(
        "Upcoming & Live Music" to BrowseFilterState.create(
            ViewTypePreset.UPCOMING_STREAMS,
            SongSegmentFilterMode.ALL,
        ),
        "Latest Streams (with segments)" to BrowseFilterState.create(
            ViewTypePreset.LATEST_STREAMS,
            SongSegmentFilterMode.REQUIRE_SONGS,
        ),
        "Latest Streams (without segments)" to BrowseFilterState.create(
            ViewTypePreset.LATEST_STREAMS,
            SongSegmentFilterMode.EXCLUDE_SONGS,
        )
    )


init {
    viewModelScope.launch {
        searchHistoryRepository.loadSearchHistory()
        viewModelScope.launch {
            browseListState
                .debounce(100L) // Add a small debounce to prevent churn during rapid upda
                .collectLatest { items ->
                    if (activeListContextType.value != MusicCategoryType.SEARCH) {
                        Timber.tag("AUTOPLAY_CONTEXT")
                            .d("BROWSE list updated. Setting context with ${items.size} it
                        continuationManager.setAutoplayContext(items)
                    }
                }
        }
        viewModelScope.launch {
            searchListState
                .debounce(100L) // Add a small debounce here as well
                .collectLatest { items ->
                    if (activeListContextType.value == MusicCategoryType.SEARCH) {
                        Timber.tag("AUTOPLAY_CONTEXT")
                            .d("SEARCH list updated. Setting context with ${items.size} it
                        continuationManager.setAutoplayContext(items)
                    }
                }
```

```
        }
    }
}

fun initializeAndFetch() {
    if (isInitialized.value) return
    isInitialized.value = true
    Timber.tag(TAG).d("ViewModel is being initialized for the first time by the UI.")

    // --- START OF MODIFICATION ---
    // REMOVE the logic that fetches organizations. It's now done automatically by the rep
    viewModelScope.launch {
        // The list is now pre-populated by the repository, so we can immediately load fil
        _browseFilterState.value = loadLastBrowseFilters()
        setupStateCollectors()
    }
    // --- END OF MODIFICATION ---
}


private fun setupStateCollectors() {
    viewModelScope.launch {
        combine(
            _activeListContextType,
            _browseFilterState.debounce(100),
            _currentSearchQuery.debounce(300)
        ) { context, filters, query -> Triple(context, filters, query) }
            .distinctUntilChanged()
            .collectLatest { (context, filters, query) ->
                if (context == MusicCategoryType.SEARCH) {
                    if (query.isNotBlank()) {
                        fetchSearchResultsInternal(
                            query,
                            isInitialLoad = true,
                            isPullToRefresh = false
                        )
                    } else {
                        clearSearchListState()
                    }
                } else {
                    fetchBrowseItemsInternal(
                        filters,
                        isInitialLoad = true,
                        isPullToRefresh = false
                    )
                }
            }
    }
}


private fun clearSearchListState() {
    rawSearchListState.apply {
        items.value = emptyList()
```

```
        isLoadingInitial.value = false
        isLoadingMore.value = false
        endOfList.value = true
        currentOffset = 0
        job?.cancel()
        job = null
        currentKeyForPrefetch = null
    }
}


// --- CREATE A NEW GENERIC HELPER FOR FETCHING ---
/**
 * A generic helper to execute a paginated fetch operation and update the corresponding st
 */
@androidx.annotation.OptIn(UnstableApi::class)
private suspend fun executePagedFetch(
    listStateHolder: ListStateHolder<HolodexVideoItem>, // <-- Now takes HolodexVideoItem
    isInitialLoad: Boolean,
    isPullToRefresh: Boolean,
    listName: String,
    fetchOperation: suspend () -> Result<FetcherResult<HolodexVideoItem>>
) {
    val mutex = if (listStateHolder == rawBrowseListState) browseMutex else searchMutex
    mutex.withLock {
        listStateHolder.job?.cancel()
        val job = viewModelScope.launch {
            if (isInitialLoad || isPullToRefresh) {
                // Only clear context for intentional, user-driven refresh operations
                if (isPullToRefresh) {
                    continuationManager.clearAutoplayContext()
                }
                listStateHolder.currentOffset = 0
                // We only clear the items visually. The context is preserved until the ne
                if (!isPullToRefresh) listStateHolder.items.value = emptyList()
                listStateHolder.endOfList.value = false
                if (isPullToRefresh) listStateHolder.isRefreshingViaPull.value = true
                else listStateHolder.isLoadingInitial.value = true
            }
            _errorMessage.value = null

            try {
                val result = fetchOperation()
                result.onSuccess { fetcherResult ->
                    // --- NO MAPPING HERE. JUST STORE THE RAW DATA ---
                    val newRawItems = fetcherResult.data

                    if (isInitialLoad || isPullToRefresh) {
                        listStateHolder.items.value = newRawItems
                    } else {
                        val currentIds = listStateHolder.items.value.map { it.id }.toSet()
                        listStateHolder.items.value += newRawItems.filter { it.id !in curr
                    }
                    listStateHolder.currentOffset += newRawItems.size
                    listStateHolder.endOfList.value =
```

```
                        newRawItems.isEmpty() || (fetcherResult.totalAvailable != null &&
                }.onFailure { exception ->
                    Timber.e(exception, "$TAG: $listName fetch FAILED.")
                    _errorMessage.value =
                        "Error fetching $listName: ${exception.localizedMessage}"
                }
            } finally {
                listStateHolder.isLoadingInitial.value = false
                listStateHolder.isRefreshingViaPull.value = false
                listStateHolder.isLoadingMore.value = false
            }
        }
        listStateHolder.job = job
    }
}


private fun fetchBrowseItemsInternal(
    filters: BrowseFilterState,
    isInitialLoad: Boolean,
    isPullToRefresh: Boolean
) {
    val offset = if (isInitialLoad || isPullToRefresh) 0 else rawBrowseListState.currentOf

    // *** THE NEW LOGIC STARTS HERE ***
    val fetchOperation: suspend () -> Result<FetcherResult<HolodexVideoItem>> = if (filter
        // --- FAVORITES FEED SPECIAL LOGIC ---
        {
            // Fetch from two sources concurrently
            val holodexFavsDeferred = viewModelScope.async {
                val favIds = holodexRepository.getFavoriteChannelIds().first()
                holodexRepository.getFavoritesFeed(favIds, filters, offset).getOrNull()?.d
            }

            val externalFavsDeferred = viewModelScope.async {
                val externalIds = localRepository.getAllExternalChannels().first()
                externalIds.flatMap { channel ->
                    holodexRepository.getMusicFromExternalChannel(channel.channelId, null)
                }
            }

            val holodexResults = holodexFavsDeferred.await()
            val externalResults = externalFavsDeferred.await()

            // Merge, sort, and paginate the combined results
            val combined = (holodexResults + externalResults)
                .distinctBy { it.id }
                .sortedByDescending { it.availableAt }

            // Manual pagination for the combined list
            val start = offset
            val end = (start + DEFAULT_PAGE_SIZE).coerceAtMost(combined.size)
            val paginatedItems = if (start >= combined.size) emptyList() else combined.sub

            Result.success(FetcherResult(paginatedItems, totalAvailable = combined.size))
```

```
        }
    } else {
        // --- STANDARD BROWSE LOGIC ---
        {
            val key = BrowseCacheKey(filters, offset)
            holodexRepository.fetchBrowseList(key, isPullToRefresh)
        }
    }

    viewModelScope.launch {
        executePagedFetch(
            rawBrowseListState,
            isInitialLoad,
            isPullToRefresh,
            "Browse",
            fetchOperation
        )
    }
}

private fun fetchSearchResultsInternal(
    query: String,
    isInitialLoad: Boolean,
    isPullToRefresh: Boolean
) {
    val offset = if (isInitialLoad || isPullToRefresh) 0 else rawSearchListState.currentOf
    val key = SearchCacheKey(query, offset)
    val fetchOperation: suspend () -> Result<FetcherResult<HolodexVideoItem>> = {
        holodexRepository.fetchSearchList(key, isPullToRefresh)
    }
    viewModelScope.launch {
        executePagedFetch(
            rawSearchListState, // Use raw holder
            isInitialLoad,
            isPullToRefresh,
            "Search",
            fetchOperation
        )
    }
}


fun loadMore(contextType: MusicCategoryType) {
    when (contextType) {
        MusicCategoryType.SEARCH -> {
            if (currentSearchQuery.value.isNotBlank()) {
                fetchSearchResultsInternal(currentSearchQuery.value, false, false)
            }
        }

        else -> fetchBrowseItemsInternal(browseFilterState.value, false, false)
    }
}

fun refreshCurrentListViaPull() {
```

```
    val context = _activeListContextType.value
    if (context == MusicCategoryType.SEARCH) {
        if (currentSearchQuery.value.isNotBlank()) {
            fetchSearchResultsInternal(currentSearchQuery.value, true, true)
        }
    } else {
        fetchBrowseItemsInternal(browseFilterState.value, true, true)
    }
}


private fun loadLastBrowseFilters(): BrowseFilterState = try {
    Gson().fromJson(
        sharedPreferences.getString(PREF_LAST_BROWSE_FILTERS, null),
        BrowseFilterState::class.java
    )
    // --- FIX: Pass the current value of the StateFlow ---
        ?: BrowseFilterState.create(
            ViewTypePreset.UPCOMING_STREAMS,
            SongSegmentFilterMode.ALL,
        )
} catch (_: Exception) {
    BrowseFilterState.create(
        ViewTypePreset.UPCOMING_STREAMS,
        SongSegmentFilterMode.ALL,
    )
}

private fun saveBrowseFilters(filters: BrowseFilterState) {
    sharedPreferences.edit { putString(PREF_LAST_BROWSE_FILTERS, Gson().toJson(filters)) }
}

private fun loadLastSearchQuery(): String =
    sharedPreferences.getString(PREF_LAST_SEARCH_QUERY, "") ?: ""

private fun saveSearchQuery(query: String) {
    sharedPreferences.edit {
        putString(
            PREF_LAST_SEARCH_QUERY, query
        )
    }; Timber.d("$TAG: Saved search query: '$query'")
}

private fun loadLastActiveListContextType(): MusicCategoryType = try {
    MusicCategoryType.valueOf(
        sharedPreferences.getString(
            PREF_LAST_CATEGORY_TYPE, MusicCategoryType.LATEST.name
        ) ?: MusicCategoryType.LATEST.name
    )
} catch (_: Exception) {
    MusicCategoryType.LATEST
}

private fun saveActiveListContextType(type: MusicCategoryType) {
    sharedPreferences.edit {
```

```
        putString(
            PREF_LAST_CATEGORY_TYPE, type.name
        )
    }; Timber.d("$TAG: Saved active context: $type")
}

fun mapVideoToPlaybackItem(video: HolodexVideoItem): PlaybackItem = PlaybackItem(
    id = video.id,
    videoId = video.id,
    songId = null,
    title = video.title,
    artistText = video.channel.name.ifBlank { video.channel.englishName ?: "" },
    albumText = video.title,
    artworkUri = video.channel.photoUrl ?: video.songs?.firstOrNull()?.artUrl,
    durationSec = video.duration,
    serverUuid = null,
    description = video.description,
    channelId = video.channel.id ?: "unknown",
    originalArtist = null
)

fun mapSongToPlaybackItem(video: HolodexVideoItem, song: HolodexSong): PlaybackItem {
    val playbackId = "${video.id}_${song.start}"
    return PlaybackItem(
        id = "${video.id}_${song.start}",
        videoId = video.id,
        songId = "${video.id}_${song.start}",
        title = song.name.ifBlank { video.title },
        artistText = video.channel.name.ifBlank { video.channel.englishName ?: "" },
        albumText = video.title,
        artworkUri = song.artUrl ?: video.channel.photoUrl,
        durationSec = (song.end - song.start).toLong().coerceAtLeast(1L),
        clipStartSec = song.start.toLong(),
        clipEndSec = song.end.toLong(),
        serverUuid = playbackId,
        description = video.description,
        channelId = video.channel.id ?: "unknown", // <-- FIX
        originalArtist = song.originalArtist
    )
}

fun setActiveListContext(type: MusicCategoryType) {
    if (_activeListContextType.value != type) {
        _activeListContextType.value = type; saveActiveListContextType(type)
    }
}

fun updateBrowseFilters(newFilters: BrowseFilterState) {
    Timber.d("$TAG: updateBrowseFilters. Old: ${_browseFilterState.value.currentFilterDisp
    if (_browseFilterState.value != newFilters) {
        _browseFilterState.value = newFilters; saveBrowseFilters(newFilters)
        if (_activeListContextType.value != MusicCategoryType.LATEST && _activeListContext
            setActiveListContext(MusicCategoryType.LATEST)
        }
    } else {
```

```
        refreshCurrentListViaPull()
    }
}

private var previousListContext: MusicCategoryType = MusicCategoryType.LATEST

fun setSearchActive(isActive: Boolean) {
    _isSearchActive.value = isActive
    if (isActive) {
        // When activating search, store the current context and switch to SEARCH mode.
        if (_activeListContextType.value != MusicCategoryType.SEARCH) {
            previousListContext = _activeListContextType.value
        }
        setActiveListContext(MusicCategoryType.SEARCH)
    }
    // NOTE: We no longer reset the context here. That's now handled by an explicit action
}

fun onSearchQueryChange(newQuery: String) {
    _currentSearchQuery.value = newQuery
}

fun performSearch(query: String) {
    val trimmedQuery = query.trim()
    if (trimmedQuery.isBlank()) {
        viewModelScope.launch { _transientMessage.emit("Please enter a search term.") }
        return
    }

    // Always clear the previous context when starting a new search
    continuationManager.clearAutoplayContext()

    // Handle special case where query is a direct video ID or URL
    extractVideoIdFromQuery(trimmedQuery)?.let { videoId ->
        navigateToVideoDetails(videoId)
        setSearchActive(false) // Close the search UI after navigating
        return
    }

    // Add query to search history
    viewModelScope.launch { searchHistoryRepository.addSearchQueryToHistory(trimmedQuery)
    _currentSearchQuery.value = trimmedQuery
    saveSearchQuery(trimmedQuery)

    // --- START OF NEW "SMART DISPATCH" LOGIC ---
    val source = _activeSearchSource.value // Assuming you have a StateFlow for the dropdo

    when (source) {
        "Holodex" -> {
            Timber.d("$TAG: Performing search on Holodex for query: '$trimmedQuery'")
            fetchSearchResultsInternal(
                query = trimmedQuery,
                isInitialLoad = true,
                isPullToRefresh = false
            )
```

```
        }
        "My Channels" -> {
            Timber.d("$TAG: Performing search on My Channels for query: '$trimmedQuery'")
            viewModelScope.launch {
                rawSearchListState.isLoadingInitial.value = true
                rawSearchListState.items.value = emptyList() // Clear previous results
                try {
                    // 1. Get the list of external channel IDs from the LocalRepository
                    val channelIds = localRepository.getAllExternalChannels().first().map
                    if (channelIds.isEmpty()) {
                        _transientMessage.emit("You haven't added any external channels ye
                        rawSearchListState.endOfList.value = true
                        return@launch
                    }

                    // 2. Call the new repository function to perform the live search
                    val result = holodexRepository.searchMusicOnChannels(trimmedQuery, cha

                    result.onSuccess { videoItems ->
                        // Update the raw search list state with the results
                        rawSearchListState.items.value = videoItems
                        rawSearchListState.endOfList.value = true // This search is not pa
                    }.onFailure { error ->
                        _errorMessage.value = "Failed to search external channels: ${error
                    }
                } finally {
                    rawSearchListState.isLoadingInitial.value = false
                }
            }
        }
        else -> {
            Timber.w("$TAG: Unknown search source selected: '$source'. Defaulting to Holod
            fetchSearchResultsInternal(
                query = trimmedQuery,
                isInitialLoad = true,
                isPullToRefresh = false
            )
        }
    }
    // --- END OF NEW "SMART DISPATCH" LOGIC ---

    // The context is already SEARCH. Now, just collapse the SearchBar UI to show the resu
    _isSearchActive.value = false
}

// You will also need to add a new StateFlow to the ViewModel for the dropdown
private val _activeSearchSource = MutableStateFlow("Holodex")
val activeSearchSource: StateFlow<String> = _activeSearchSource.asStateFlow()

fun setActiveSearchSource(source: String) {
    _activeSearchSource.value = source
}

// NEW: The explicit action to handle user cancellation of a search.
fun clearSearchAndReturnToBrowse() {
```

```
    _currentSearchQuery.value = ""
    saveSearchQuery("")
    setActiveListContext(previousListContext) // Revert to the context before search was a
    _isSearchActive.value = false // Ensure the search bar UI is collapsed
}

fun performSearchByChannelId(channelId: String) {
    val newQuery = "$CHANNEL_ID_SEARCH_PREFIX$channelId"
    _currentSearchQuery.value = newQuery
    saveSearchQuery(newQuery)
    setActiveListContext(MusicCategoryType.SEARCH)
    _navigationRequest.value = NavigationDestination.HomeScreenWithSearch
}

fun requestPlaybackForPreparedList(
    items: List<PlaybackItem>,
    startIndex: Int = 0,
    startPositionSec: Long = 0L,
    shouldShuffle: Boolean = false
) {
    if (items.isEmpty()) {
        _errorMessage.value = "Cannot play empty list."; return
    }
    viewModelScope.launch {
        playbackRequestManager.submitPlaybackRequest(
            items,
            startIndex,
            startPositionSec,
            shouldShuffle
        )
    }
}

fun prepareAndRequestPlayback(
    videoItem: HolodexVideoItem, songToStartWith: HolodexSong? = null
) {
    val itemToPlay =
        songToStartWith?.let { mapSongToPlaybackItem(videoItem, it) } ?: mapVideoToPlaybac
            videoItem
        )
    requestPlaybackForPreparedList(listOf(itemToPlay))
}


fun navigateToVideoDetails(videoId: String) {
    _navigationRequest.value = NavigationDestination.VideoDetails(videoId)
}

// Action Handlers

fun onVideoClicked(item: UnifiedDisplayItem) {
    viewModelScope.launch {
        // Find the full HolodexVideoItem from the raw list
        val rawItem = rawBrowseListState.items.value.find { it.id == item.videoId }
            ?: rawSearchListState.items.value.find { it.id == item.videoId }
```

```
        if (rawItem != null) {
            _videoItemForDetailScreen.value = rawItem
            navigateToVideoDetails(item.videoId)
        } else {
            // Fallback for cases where the raw item isn't in the current list
            // (e.g., navigating from a different screen)
            _videoItemForDetailScreen.value = null // Ensure no stale data is used
            navigateToVideoDetails(item.videoId)
        }
    }
}


// This function is now simplified as it only needs to convert from the universal item
fun playFavoriteOrLikedSegmentItem(item: PlaybackItem) {
    viewModelScope.launch { requestPlaybackForPreparedList(listOf(item)) }
}


fun clearNavigationRequest() {
    _navigationRequest.value = null
}

fun clearErrorMessage() {
    _errorMessage.value = null
}

fun setBrowseContextAndNavigate(
    org: String? = null,
    channelId: String? = null
) {
    Timber.d("Setting new browse context. Org: $org, ChannelId: $channelId")
    // If a channelId is provided, we must use the SEARCH context.
    if (channelId != null) {
        val newQuery = "$CHANNEL_ID_SEARCH_PREFIX$channelId"
        onSearchQueryChange(newQuery) // Update the query text
        saveSearchQuery(newQuery)
        setActiveListContext(MusicCategoryType.SEARCH)
        // CRUCIAL: Do NOT activate the search bar UI.
        _isSearchActive.value = false
    }
    // If only an org is provided, we use the BROWSE context with a filter.
    else {
        val newFilter = BrowseFilterState.create(
            preset = ViewTypePreset.LATEST_STREAMS,
            songFilterMode = SongSegmentFilterMode.REQUIRE_SONGS,
            organization = org?.takeIf { it != "All Vtubers" },
        )
        updateBrowseFilters(newFilter)
        setActiveListContext(MusicCategoryType.LATEST)
    }
}

fun addVideoOrItsSegmentsToQueue(item: PlaybackItem) {
```

```
        viewModelScope.launch {
            _isLoadingGeneral.value = true
            addOrFetchAndAddUseCase(item)
                .onSuccess { message ->
                    _transientMessage.emit(message) // <-- FIX: Use emit on the SharedFlow
                }
                .onFailure { error ->
                    _errorMessage.value = "Failed to add to queue: ${error.localizedMessage}"
                }
            _isLoadingGeneral.value = false
        }
    }
}

// File: java\com\example\holodex\viewmodel\autoplay\AutoplayItemProvider.kt
package com.example.holodex.viewmodel.autoplay

import com.example.holodex.data.repository.HolodexRepository
import com.example.holodex.playback.domain.model.PlaybackItem
import com.example.holodex.viewmodel.UnifiedDisplayItem
import com.example.holodex.data.model.HolodexSong
import com.example.holodex.data.model.HolodexVideoItem
import com.example.holodex.viewmodel.mappers.toPlaybackItem
import timber.log.Timber

internal sealed class AutoplaySearchResult {
    data class Found(val item: UnifiedDisplayItem) : AutoplaySearchResult()
    data class NotFound(val reason: String) : AutoplaySearchResult()
}

class AutoplayItemProvider(
    private val holodexRepository: HolodexRepository
) {
    companion object {
        private const val TAG = "AutoplayItemProvider"
    }

    suspend fun provideNextItemsForAutoplay(
        currentScreenItems: List<UnifiedDisplayItem>,
        lastPlayedItemIdInQueue: String?,
        unifiedDisplayItemToPlaybackItem: (UnifiedDisplayItem) -> PlaybackItem,
        holodexSongToPlaybackItem: (HolodexVideoItem, HolodexSong) -> PlaybackItem
    ): List<PlaybackItem>? {
        if (currentScreenItems.isEmpty()) {
            Timber.w("$TAG: Current screen list is empty.")
            return null
        }

        try {
            val nextCandidateResult = findNextCandidate(currentScreenItems, lastPlayedItemIdIn

            if (nextCandidateResult is AutoplaySearchResult.Found) {
                val candidateItem = nextCandidateResult.item
                Timber.i("$TAG: Found next autoplay candidate: '${candidateItem.title}'")
```

```
            if (shouldCheckForSegments(candidateItem)) {
                val videoWithSongsResult = holodexRepository.getVideoWithSongs(candidateIt
                if (videoWithSongsResult.isSuccess) {
                    val videoWithSongs = videoWithSongsResult.getOrThrow()
                    if (!videoWithSongs.songs.isNullOrEmpty()) {
                        Timber.i("$TAG: Video '${candidateItem.title}' has ${videoWithSong
                        return videoWithSongs.songs.map { song ->
                            holodexSongToPlaybackItem(videoWithSongs, song)
                        }
                    }
                }
            }

            // If not checking for segments or if it fails, play the single item
            val singlePlaybackItem = unifiedDisplayItemToPlaybackItem(candidateItem)
            if (validateAutoplayItem(singlePlaybackItem, lastPlayedItemIdInQueue)) {
                return listOf(singlePlaybackItem)
            }
            return null // Loop prevented
        } else {
            val reason = (nextCandidateResult as AutoplaySearchResult.NotFound).reason
            Timber.i("$TAG: No next autoplay candidate found. Reason: $reason.")
            return null
        }

    } catch (e: Exception) {
        Timber.e(e, "$TAG: Exception during autoplay provider execution.")
        return null
    }
}

private fun findNextCandidate(
    currentScreenItems: List<UnifiedDisplayItem>,
    lastPlayedItemIdInQueue: String?
): AutoplaySearchResult {
    if (lastPlayedItemIdInQueue == null) {
        return if (currentScreenItems.isNotEmpty()) {
            AutoplaySearchResult.Found(currentScreenItems.first())
        } else {
            AutoplaySearchResult.NotFound("screen_list_empty")
        }
    }

    val lastUnderscoreIndex = lastPlayedItemIdInQueue.lastIndexOf('_')

    // Check if there's an underscore and if the part after it is a number (the start time
    val lastPartIsNumeric = lastUnderscoreIndex != -1 &&
            lastPlayedItemIdInQueue.substring(lastUnderscoreIndex + 1).all { it.isDigit()

    val lastPlayedVideoId = if (lastPartIsNumeric) {
        // It's a song segment ID, so take the part before the last underscore.
        lastPlayedItemIdInQueue.substring(0, lastUnderscoreIndex)
    } else {
        // It's just a video ID, so use the whole string.
        lastPlayedItemIdInQueue
```

```
        }

        val indexOfLastPlayed = currentScreenItems.indexOfFirst { it.videoId == lastPlayedVide

        if (indexOfLastPlayed != -1 && indexOfLastPlayed + 1 < currentScreenItems.size) {
            return AutoplaySearchResult.Found(currentScreenItems[indexOfLastPlayed + 1])
        }

        return AutoplaySearchResult.NotFound("end_of_list_reached")
    }


    private fun shouldCheckForSegments(item: UnifiedDisplayItem): Boolean {
        return !item.isSegment && (item.songCount ?: 0) > 0
    }

    private fun validateAutoplayItem(
        itemToPlay: PlaybackItem?,
        lastPlayedItemIdInQueue: String?
    ): Boolean {
        if (itemToPlay == null) return false
        if (lastPlayedItemIdInQueue == null) return true
        val notLooping = itemToPlay.id != lastPlayedItemIdInQueue
        if (!notLooping) {
            Timber.w("$TAG: Loop prevention! Attempted to autoplay same item ID: ${itemToPlay.
        }
        return notLooping
    }
}

// File: java\com\example\holodex\viewmodel\autoplay\ContinuationManager.kt
// File: java/com/example/holodex/viewmodel/autoplay/ContinuationManager.kt
// (Create this new file)

package com.example.holodex.viewmodel.autoplay

import com.example.holodex.data.repository.HolodexRepository
import com.example.holodex.data.repository.UserPreferencesRepository
import com.example.holodex.playback.domain.model.PlaybackItem
import com.example.holodex.playback.domain.model.PlaybackQueue
import com.example.holodex.playback.domain.repository.PlaybackRepository
import com.example.holodex.viewmodel.UnifiedDisplayItem
import com.example.holodex.viewmodel.mappers.toPlaybackItem
import com.example.holodex.viewmodel.mappers.toVideoShell
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.Job
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.flow.collectLatest
import kotlinx.coroutines.flow.first
import kotlinx.coroutines.launch
import timber.log.Timber
import java.util.Collections
```

```
import javax.inject.Inject
import javax.inject.Singleton

@Singleton
class ContinuationManager @Inject constructor(
    private val holodexRepository: HolodexRepository,
    private val userPreferencesRepository: UserPreferencesRepository,
    private val autoplayItemProvider: AutoplayItemProvider
) {

    companion object {
        private const val TAG = "ContinuationManager"
        private const val RADIO_QUEUE_THRESHOLD = 5
    }

    private var autoplayContextItems: List<UnifiedDisplayItem> = Collections.synchronizedList(

    private val _isRadioModeActive = MutableStateFlow(false)
    val isRadioModeActive: StateFlow<Boolean> = _isRadioModeActive.asStateFlow()

    private var currentRadioId: String? = null
    private var radioMonitorJob: Job? = null

    /**
     * Called by ViewModels to provide the current list of items on screen,
     * which will be used as the source for autoplay suggestions.
     */
    fun setAutoplayContext(items: List<UnifiedDisplayItem>) {
        // Only update context if the new list is not empty.
        // This prevents transient empty states from wiping a valid, existing context.
        if (items.isNotEmpty()) {
            Timber.d("$TAG: Setting autoplay context with ${items.size} items.")
            autoplayContextItems = Collections.synchronizedList(items.toMutableList())
        } else {
            Timber.d("$TAG: Ignoring empty context update. Keeping existing context with ${aut
        }
    }

    /**
     * Explicit method for intentionally clearing the autoplay context when a user
     * performs an action that should reset it, like a new search or pull-to-refresh.
     */
    fun clearAutoplayContext() {
        Timber.d("$TAG: Explicitly clearing autoplay context.")
        autoplayContextItems = Collections.synchronizedList(mutableListOf())
    }

    /**
     * Starts a new radio session. Fetches the initial batch of songs and begins monitoring th
     * @return The initial list of PlaybackItems to start the radio.
     */
    suspend fun startRadioSession(radioId: String, scope: CoroutineScope, playbackRepository:
        endCurrentSession() // Stop any previous session
        currentRadioId = radioId
        _isRadioModeActive.value = true
```

```
    Timber.d("$TAG: Starting radio session for ID: $radioId")

    val initialBatch = fetchRadioBatch(radioId)
    if (initialBatch.isNullOrEmpty()) {
        Timber.e("$TAG: Failed to fetch initial batch for radio $radioId. Aborting session
        endCurrentSession()
        return null
    }

    // Start the job that will monitor the queue
    radioMonitorJob = scope.launch(Dispatchers.IO) {
        Timber.tag(TAG).i("RADIO_LOG: Monitor job LAUNCHED for radio: $radioId")
        playbackRepository.observePlaybackQueue().collectLatest { queue ->
            handleQueueStateForRadio(queue, playbackRepository)
        }
    }
    radioMonitorJob?.invokeOnCompletion {
        Timber.tag(TAG).i("RADIO_LOG: Monitor job COMPLETED/CANCELLED for radio: $radioId"
    }

    return initialBatch
}

/**
 * Ends the current radio session, stopping any background monitoring.
 */
fun endCurrentSession() {
    if (currentRadioId != null) {
        Timber.d("$TAG: Ending radio session for ID: $currentRadioId")
    }
    radioMonitorJob?.cancel()
    radioMonitorJob = null
    currentRadioId = null
    _isRadioModeActive.value = false
}

/**
 * Provides the next items to play when a finite queue ends, respecting the user's autopla
 * @return A list of new PlaybackItems to append, or null if autoplay is disabled or no it
 */
suspend fun provideAutoplayItems(currentQueue: List<PlaybackItem>): List<PlaybackItem>? {
    val isAutoplayEnabled = userPreferencesRepository.autoplayEnabled.first()
    if (!isAutoplayEnabled) {
        Timber.d("$TAG: Autoplay is disabled by user setting. Not providing items.")
        return null
    }

    Timber.d("$TAG: Autoplay is enabled. Attempting to provide next items.")


    return autoplayItemProvider.provideNextItemsForAutoplay(
        currentScreenItems = autoplayContextItems,
        lastPlayedItemIdInQueue = currentQueue.lastOrNull()?.id,
        { item -> item.toPlaybackItem() }, // Pass the mapper function
        { video, song -> song.toPlaybackItem(video) } // Pass the other mapper function
```

```
        )
    }

    private suspend fun handleQueueStateForRadio(queue: PlaybackQueue, playbackRepository: Pla
        val radioId = currentRadioId ?: return // Session ended

        if (radioId == null) {
            Timber.tag(TAG).d("RADIO_LOG: handleQueueState called but no active radio session.
            return
        }
        val songsRemaining = queue.items.size - (queue.currentIndex + 1)
        Timber.tag(TAG).d("RADIO_LOG: Queue state update. Songs: ${queue.items.size}, Index: $

        if (songsRemaining < RADIO_QUEUE_THRESHOLD) {
            Timber.d("$TAG: Radio queue threshold reached ($songsRemaining remaining). Fetchin
            val nextBatch = fetchRadioBatch(radioId)
            if (!nextBatch.isNullOrEmpty()) {
                playbackRepository.addItemsToQueue(nextBatch)
                Timber.d("$TAG: Appended ${nextBatch.size} new songs to the radio queue.")
            } else {
                Timber.w("$TAG: Fetching next radio batch returned no items.")
            }
        }
    }

    private suspend fun fetchRadioBatch(radioId: String): List<PlaybackItem>? {
        val result = holodexRepository.getRadioContent(radioId)
        return result.getOrNull()?.content?.mapNotNull { song ->
            val videoShell = song.toVideoShell(result.getOrNull()?.title ?: "")
            song.toPlaybackItem(videoShell)
        }
    }
}

// File: java\com\example\holodex\viewmodel\mappers\UnifiedDisplayItemMapper.kt
// File: java/com/example/holodex/viewmodel/mappers/UnifiedDisplayItemMapper.kt
package com.example.holodex.viewmodel.mappers

import com.example.holodex.data.db.DownloadedItemEntity
import com.example.holodex.data.db.HistoryItemEntity
import com.example.holodex.data.db.LikedItemEntity
import com.example.holodex.data.db.LikedItemType
import com.example.holodex.data.db.PlaylistItemEntity
import com.example.holodex.data.model.HolodexChannelMin
import com.example.holodex.data.model.HolodexSong
import com.example.holodex.data.model.HolodexVideoItem
import com.example.holodex.data.model.discovery.MusicdexSong
import com.example.holodex.playback.domain.model.PlaybackItem
import com.example.holodex.playback.util.formatDurationSeconds
import com.example.holodex.util.ThumbnailQuality
import com.example.holodex.util.generateArtworkUrlList
import com.example.holodex.viewmodel.UnifiedDisplayItem
import kotlin.math.max

// Mappers to UnifiedDisplayItem
```

```
fun PlaylistItemEntity.toUnifiedDisplayItem(
    isDownloaded: Boolean,
    isLiked: Boolean
): UnifiedDisplayItem {
    val isSegment = this.itemTypeInPlaylist == LikedItemType.SONG_SEGMENT
    val durationSec = if (isSegment && this.songStartSecondsPlaylist != null && this.songEndSe
        max(1, (this.songEndSecondsPlaylist - this.songStartSecondsPlaylist)).toLong()
    } else {
        0L
    }

    return UnifiedDisplayItem(
        stableId = "playlist_${this.playlistOwnerId}_${this.itemIdInPlaylist}",
        playbackItemId = this.itemIdInPlaylist,
        videoId = this.videoIdForItem,
        channelId = "", // This info is not stored in PlaylistItemEntity, might need to be add
        title = this.songNamePlaylist ?: "Unknown Title",
        artistText = this.songArtistTextPlaylist ?: "Unknown Artist",
        artworkUrls = listOfNotNull(this.songArtworkUrlPlaylist),
        durationText = formatDurationSeconds(durationSec),
        isSegment = isSegment,
        songCount = null,
        isDownloaded = isDownloaded,
        isLiked = isLiked,
        itemTypeForPlaylist = this.itemTypeInPlaylist,
        songStartSec = this.songStartSecondsPlaylist,
        songEndSec = this.songEndSecondsPlaylist,
        originalArtist = this.songArtistTextPlaylist,
        isExternal = this.isLocalOnly
    )
}

fun HolodexVideoItem.toUnifiedDisplayItem(
    isLiked: Boolean,
    downloadedSegmentIds: Set<String>
): UnifiedDisplayItem {
    val containsDownloadedSegments = this.songs?.any { song ->
        val segmentId = "${this.id}_${song.start}"
        downloadedSegmentIds.contains(segmentId)
    } == true

    return UnifiedDisplayItem(
        stableId = "video_${this.id}",
        playbackItemId = this.id,
        videoId = this.id,
        channelId = this.channel.id ?: this.id,
        title = this.title,
        artistText = this.channel.name,
        artworkUrls = generateArtworkUrlList(this.toPlaybackItem(), ThumbnailQuality.MEDIUM),
        durationText = formatDurationSeconds(this.duration),
        isSegment = false,
        songCount = this.songcount,
        isDownloaded = containsDownloadedSegments,
        isLiked = isLiked,
```

```
        itemTypeForPlaylist = LikedItemType.VIDEO,
        songStartSec = null,
        songEndSec = null,
        originalArtist = null,
        isExternal = this.channel.org == "External"
    )
}

fun HolodexSong.toUnifiedDisplayItem(
    parentVideo: HolodexVideoItem,
    isLiked: Boolean,
    isDownloaded: Boolean
): UnifiedDisplayItem {
    val playbackItemId = "${parentVideo.id}_${this.start}"
    return UnifiedDisplayItem(
        stableId = "song_${playbackItemId}",
        playbackItemId = playbackItemId,
        videoId = parentVideo.id,
        channelId = parentVideo.channel.id ?: parentVideo.id,
        title = this.name,
        artistText = parentVideo.channel.name,
        artworkUrls = generateArtworkUrlList(this.toPlaybackItem(parentVideo), ThumbnailQualit
        durationText = formatDurationSeconds((this.end - this.start).toLong()),
        isSegment = true,
        songCount = null,
        isDownloaded = isDownloaded,
        isLiked = isLiked,
        itemTypeForPlaylist = LikedItemType.SONG_SEGMENT,
        songStartSec = this.start,
        songEndSec = this.end,
        originalArtist = this.originalArtist,
        isExternal = parentVideo.channel.org == "External"
    )
}

fun LikedItemEntity.toUnifiedDisplayItem(isDownloaded: Boolean): UnifiedDisplayItem {
    val isSegment = this.itemType == LikedItemType.SONG_SEGMENT
    return UnifiedDisplayItem(
        stableId = "liked_${this.itemId}",
        playbackItemId = this.itemId,
        videoId = this.videoId,
        channelId = this.channelIdSnapshot,
        title = if (isSegment) this.actualSongName ?: this.titleSnapshot else this.titleSnapsh
        artistText = if (isSegment) this.actualSongArtist ?: this.artistTextSnapshot else this
        artworkUrls = generateArtworkUrlList(this.toPlaybackItem(), ThumbnailQuality.MEDIUM),
        durationText = formatDurationSeconds(this.durationSecSnapshot),
        isSegment = isSegment,
        songCount = null,
        isDownloaded = isDownloaded,
        isLiked = true,
        itemTypeForPlaylist = this.itemType,
        songStartSec = this.songStartSeconds,
        songEndSec = this.songEndSeconds,
        originalArtist = this.actualSongArtist,
        isExternal = false
```

```
    )
}

fun HistoryItemEntity.toUnifiedDisplayItem(
    isDownloaded: Boolean,
    isLiked: Boolean
): UnifiedDisplayItem {
    return UnifiedDisplayItem(
        stableId = "history_${this.playedAtTimestamp}",
        playbackItemId = this.itemId,
        videoId = this.videoId,
        channelId = this.channelId,
        title = this.title,
        artistText = this.artistText,
        artworkUrls = generateArtworkUrlList(this.toPlaybackItem(), ThumbnailQuality.MEDIUM),
        durationText = formatDurationSeconds(this.durationSec),
        isSegment = true,
        songCount = null,
        isDownloaded = isDownloaded,
        isLiked = isLiked,
        itemTypeForPlaylist = LikedItemType.SONG_SEGMENT,
        songStartSec = this.songStartSeconds,
        songEndSec = (this.songStartSeconds + this.durationSec).toInt(),
        originalArtist = null,
        isExternal = false
    )
}

fun DownloadedItemEntity.toUnifiedDisplayItem(isLiked: Boolean): UnifiedDisplayItem {
    val parentVideoId = this.videoId.substringBeforeLast('_')
    val songStartSec = this.videoId.substringAfterLast('_').toIntOrNull()

    return UnifiedDisplayItem(
        stableId = "download_${this.videoId}",
        playbackItemId = this.videoId,
        videoId = parentVideoId,
        channelId = this.channelId,
        title = this.title,
        artistText = this.artistText,
        artworkUrls = generateArtworkUrlList(this.toPlaybackItem(), ThumbnailQuality.MEDIUM),
        durationText = formatDurationSeconds(this.durationSec),
        isSegment = true,
        songCount = null,
        isDownloaded = true,
        isLiked = isLiked,
        itemTypeForPlaylist = LikedItemType.SONG_SEGMENT,
        songStartSec = songStartSec,
        songEndSec = songStartSec?.let { it + this.durationSec.toInt() },
        originalArtist = this.artistText,
        isExternal = false
    )
}

// Mappers to PlaybackItem
```

```
fun UnifiedDisplayItem.toPlaybackItem(): PlaybackItem {
    return PlaybackItem(
        id = this.playbackItemId,
        videoId = this.videoId,
        serverUuid = if (this.isSegment) this.playbackItemId else null,
        songId = if (this.isSegment) this.playbackItemId else null,
        title = this.title,
        artistText = this.artistText,
        albumText = if (!this.isSegment) this.title else null,
        artworkUri = this.artworkUrls.firstOrNull(),
        durationSec = this.songEndSec?.toLong()?.let { it - (this.songStartSec?.toLong() ?: 0)
        streamUri = null,
        clipStartSec = this.songStartSec?.toLong(),
        clipEndSec = this.songEndSec?.toLong(),
        description = null,
        channelId = this.channelId,
        originalArtist = this.originalArtist,
        isExternal = this.isExternal
    )
}

fun HolodexVideoItem.toPlaybackItem(): PlaybackItem {
    return PlaybackItem(
        id = this.id,
        videoId = this.id,
        serverUuid = null,
        songId = null,
        title = this.title,
        artistText = this.channel.name,
        albumText = this.title,
        artworkUri = this.channel.photoUrl,
        durationSec = this.duration,
        streamUri = null,
        clipStartSec = null,
        clipEndSec = null,
        description = this.description,
        channelId = this.channel.id ?: "unknown_channel_${this.id}",
        originalArtist = null,
        isExternal = this.channel.org == "External"
    )
}

internal fun HolodexSong.toPlaybackItem(parentVideo: HolodexVideoItem): PlaybackItem {
    val playbackId = "${parentVideo.id}_${this.start}"
    return PlaybackItem(
        id = playbackId,
        videoId = parentVideo.id,
        serverUuid = playbackId,
        songId = playbackId,
        title = this.name,
        artistText = parentVideo.channel.name,
        albumText = parentVideo.title,
        artworkUri = this.artUrl,
        durationSec = (this.end - this.start).toLong(),
        streamUri = null,
```

```
        clipStartSec = this.start.toLong(),
        clipEndSec = this.end.toLong(),
        description = parentVideo.description,
        channelId = parentVideo.channel.id ?: "unknown_channel_${parentVideo.id}",
        originalArtist = this.originalArtist,
        isExternal = parentVideo.channel.org == "External"
    )
}

internal fun LikedItemEntity.toPlaybackItem(): PlaybackItem {
    return PlaybackItem(
        id = this.itemId,
        videoId = this.videoId,
        serverUuid = this.serverId,
        songId = if (this.itemType == LikedItemType.SONG_SEGMENT) this.itemId else null,
        title = this.titleSnapshot,
        artistText = this.artistTextSnapshot,
        albumText = this.albumTextSnapshot,
        artworkUri = this.artworkUrlSnapshot,
        durationSec = this.durationSecSnapshot,
        streamUri = null,
        clipStartSec = this.songStartSeconds?.toLong(),
        clipEndSec = this.songEndSeconds?.toLong(),
        description = this.descriptionSnapshot,
        channelId = this.channelIdSnapshot,
        originalArtist = this.actualSongArtist,
        isExternal = false
    )
}

internal fun HistoryItemEntity.toPlaybackItem(): PlaybackItem {
    return PlaybackItem(
        id = this.itemId,
        videoId = this.videoId,
        serverUuid = null,
        songId = this.itemId,
        title = this.title,
        artistText = this.artistText,
        albumText = this.title,
        artworkUri = this.artworkUrl,
        durationSec = this.durationSec,
        streamUri = null,
        clipStartSec = this.songStartSeconds.toLong(),
        clipEndSec = (this.songStartSeconds + this.durationSec),
        description = null,
        channelId = this.channelId,
        originalArtist = null,
        isExternal = false
    )
}

internal fun DownloadedItemEntity.toPlaybackItem(): PlaybackItem {
    val parentVideoId = this.videoId.substringBeforeLast('_')
    val songStartSec = this.videoId.substringAfterLast('_').toLongOrNull()
```

```
    return PlaybackItem(
        id = this.videoId,
        videoId = parentVideoId,
        serverUuid = this.videoId,
        songId = this.videoId,
        title = this.title,
        artistText = this.artistText,
        albumText = this.title,
        artworkUri = this.artworkUrl,
        durationSec = this.durationSec,
        streamUri = this.localFileUri,
        clipStartSec = songStartSec,
        clipEndSec = songStartSec?.let { it + this.durationSec },
        description = null,
        channelId = this.channelId,
        originalArtist = this.artistText,
        isExternal = false
    )
}

internal fun MusicdexSong.toVideoShell(): HolodexVideoItem {
    return HolodexVideoItem(
        id = this.videoId, title = "Unknown Video", type = "stream", topicId = null,
        availableAt = "", publishedAt = null, duration = (this.end - this.start).toLong(),
        status = "past",
        channel = HolodexChannelMin(
            id = this.channel.id ?: this.channelId, name = this.channel.name,
            englishName = this.channel.englishName, org = null, type = "vtuber",
            photoUrl = this.channel.photoUrl
        ),
        songcount = 1, description = null, songs = null
    )
}
internal fun MusicdexSong.toVideoShell(albumTitle: String): HolodexVideoItem {
    return this.toVideoShell().copy(title = albumTitle)
}
fun MusicdexSong.toUnifiedDisplayItem(
    parentVideo: HolodexVideoItem,
    isLiked: Boolean,
    isDownloaded: Boolean
): UnifiedDisplayItem {
    val playbackItemId = "${this.videoId}_${this.start}"
    return UnifiedDisplayItem(
        stableId = "song_${playbackItemId}", playbackItemId = playbackItemId, videoId = this.v
        channelId = this.channel.id ?: "", title = this.name, artistText = this.channel.name,
        artworkUrls = generateArtworkUrlList(this.toPlaybackItem(parentVideo), ThumbnailQualit
        durationText = formatDurationSeconds((this.end - this.start).toLong()),
        isSegment = true, songCount = null, isDownloaded = isDownloaded, isLiked = isLiked,
        itemTypeForPlaylist = LikedItemType.SONG_SEGMENT, songStartSec = this.start,
        songEndSec = this.end, originalArtist = this.originalArtist,
        isExternal = parentVideo.channel.org == "External"
    )
}

internal fun MusicdexSong.toPlaybackItem(parentVideo: HolodexVideoItem): PlaybackItem {
```

```
    val reliableChannelId = this.channelId ?: parentVideo.channel.id ?: this.videoId
    val playbackId = "${this.videoId}_${this.start}"

    return PlaybackItem(
        id = playbackId,
        videoId = this.videoId,
        serverUuid = this.id,
        songId = playbackId,
        title = this.name,
        artistText = this.channel.name,
        albumText = parentVideo.title,
        artworkUri = this.artUrl,
        durationSec = (this.end - this.start).toLong(),
        streamUri = null,
        clipStartSec = this.start.toLong(),
        clipEndSec = this.end.toLong(),
        description = null,
        channelId = reliableChannelId,
        originalArtist = this.originalArtist,
        isExternal = false
    )
}

fun HolodexVideoItem.toVirtualSegmentUnifiedDisplayItem(
    isLiked: Boolean,
    isDownloaded: Boolean
): UnifiedDisplayItem {
    val playbackItemId = "${this.id}_0"
    return UnifiedDisplayItem(
        stableId = "video_as_segment_${this.id}",
        playbackItemId = playbackItemId,
        videoId = this.id,
        channelId = this.channel.id ?: "",
        title = this.title,
        artistText = this.channel.name,
        artworkUrls = generateArtworkUrlList(this.toPlaybackItem(), ThumbnailQuality.MEDIUM),
        durationText = formatDurationSeconds(this.duration),
        isSegment = true,
        songCount = 0,
        isDownloaded = isDownloaded,
        isLiked = isLiked,
        itemTypeForPlaylist = LikedItemType.VIDEO,
        songStartSec = 0,
        songEndSec = this.duration.toInt(),
        originalArtist = null,
        isExternal = this.channel.org == "External"
    )
}

// File: java\com\example\holodex\viewmodel\state\BrowseFilterState.kt
package com.example.holodex.viewmodel.state

// Enum for the main view type, which dictates API parameters like 'status'
enum class ViewTypePreset(
    val apiStatus: String?, // e.g., "past", "upcoming"
```

```
    val apiMaxUpcomingHours: Int?,
    val defaultSortField: VideoSortField,
    val defaultSortOrder: SortOrder,
    val defaultDisplayName: String // Base display name for this preset
) {
    LATEST_STREAMS(
        apiStatus = "past",
        apiMaxUpcomingHours = null,
        defaultSortField = VideoSortField.AVAILABLE_AT,
        defaultSortOrder = SortOrder.DESC,
        defaultDisplayName = "Latest Music Streams"
    ),
    UPCOMING_STREAMS(
        apiStatus = "upcoming",
        apiMaxUpcomingHours = 48,
        defaultSortField = VideoSortField.START_SCHEDULED,
        defaultSortOrder = SortOrder.ASC,
        defaultDisplayName = "Upcoming Music (Next 48h)"
    );
}

// Enum for client-side filtering based on song segments
enum class SongSegmentFilterMode(val displayNameSuffix: String?) {
    ALL(null), // Show all (after initial API and music content filtering)
    REQUIRE_SONGS(" (with segments)"),
    EXCLUDE_SONGS(" (without segments)");
}

// Enum for API sort fields
enum class VideoSortField(val apiValue: String, val displayName: String) {
    AVAILABLE_AT("available_at", "Date"),
    PUBLISHED_AT("published_at", "Published Date"),
    START_SCHEDULED("start_scheduled", "Scheduled Time"),
    START_ACTUAL("start_actual", "Actual Start"),
    DURATION("duration", "Duration"),
    LIVE_VIEWERS("live_viewers", "Live Viewers"),
    SONG_COUNT("songcount", "Song Count"),
    TITLE("title", "Title")
}

// Enum for API sort order
enum class SortOrder(val apiValue: String, val displayName: String) {
    ASC("asc", "Ascending"),
    DESC("desc", "Descending")
}

data class BrowseFilterState(
    val selectedOrganization: String? = null, // API value of the org (e.g., "Hololive")
    val selectedPrimaryTopic: String? = null, // API value of the topic (e.g., "singing")

    val selectedViewPreset: ViewTypePreset,
    val songSegmentFilterMode: SongSegmentFilterMode, // Client-side filter for LATEST_STREAMS

    val sortField: VideoSortField,
    val sortOrder: SortOrder,
```

```
    val currentFilterDisplayName: String
) {
    // Computed properties for API parameters, derived from selectedViewPreset
    val status: String? get() = selectedViewPreset.apiStatus
    val maxUpcomingHours: Int? get() = selectedViewPreset.apiMaxUpcomingHours

    companion object {

        fun create(
            preset: ViewTypePreset,
            songFilterMode: SongSegmentFilterMode,
            organization: String? = null,
            primaryTopic: String? = null,
            sortFieldOverride: VideoSortField? = null,
            sortOrderOverride: SortOrder? = null
        ): BrowseFilterState {
            val effectiveSortField = sortFieldOverride ?: preset.defaultSortField
            val effectiveSortOrder = sortOrderOverride ?: preset.defaultSortOrder

            // The display name will now be generated inside the Composable.
            // We store a placeholder or base name here.
            val displayName = preset.defaultDisplayName

            return BrowseFilterState(
                selectedOrganization = organization,
                selectedPrimaryTopic = primaryTopic,
                selectedViewPreset = preset,
                songSegmentFilterMode = songFilterMode,
                sortField = effectiveSortField,
                sortOrder = effectiveSortOrder,
                currentFilterDisplayName = displayName // This will be updated by the UI
            )
        }
        // --- END OF MODIFICATION ---
    }

    // Helper to check if any non-default filters are active
    @get:JvmName("getHasActiveFiltersProperty")
    val hasActiveFilters: Boolean
        get() {
            val defaultForPreset = create(
                preset = this.selectedViewPreset,
                songFilterMode = SongSegmentFilterMode.ALL, // Default song filter for compari
                organization = null,
                primaryTopic = null,
                sortFieldOverride = this.selectedViewPreset.defaultSortField,
                sortOrderOverride = this.selectedViewPreset.defaultSortOrder,
            )

            return this.selectedOrganization != defaultForPreset.selectedOrganization ||
                    this.selectedPrimaryTopic != defaultForPreset.selectedPrimaryTopic ||
                    this.sortField != defaultForPreset.sortField ||
                    this.sortOrder != defaultForPreset.sortOrder ||
                    (this.selectedViewPreset == ViewTypePreset.LATEST_STREAMS && this.songSegm
```

```
                    this.selectedViewPreset != ViewTypePreset.LATEST_STREAMS // if it's not th
        }

    // Helper to count active filters for badge
    @get:JvmName("getActiveFilterCountProperty")
    val activeFilterCount: Int
        get() {
            var count = 0
            val defaultForPreset = create( // Create a default state for the current preset fo
                preset = this.selectedViewPreset,
                songFilterMode = SongSegmentFilterMode.ALL,
                organization = null,
                primaryTopic = null,
                sortFieldOverride = this.selectedViewPreset.defaultSortField,
                sortOrderOverride = this.selectedViewPreset.defaultSortOrder,
            )

            if (this.selectedOrganization != defaultForPreset.selectedOrganization) count++
            if (this.selectedPrimaryTopic != defaultForPreset.selectedPrimaryTopic) count++ //
            if (this.sortField != defaultForPreset.sortField || this.sortOrder != defaultForPr

            // Count song segment filter only if it's for LATEST_STREAMS and not ALL
            if (this.selectedViewPreset == ViewTypePreset.LATEST_STREAMS && this.songSegmentFi
                count++
            }
            // If the view preset itself is not the "default" (e.g., LATEST_STREAMS with ALL s
            // This logic might need adjustment based on what you consider a "base default".
            // For now, let's say changing the view preset is a filter.
            if (this.selectedViewPreset != ViewTypePreset.LATEST_STREAMS) { // Assuming LATEST
                count++
            } else if (this.selectedViewPreset == ViewTypePreset.LATEST_STREAMS && this.songSe
                // if it IS latest streams, but song filter is active, it's already counted.
                // This else-if prevents double counting if the default view is LATEST + ALL a
            }


            return count
        }
}

// File: java\com\example\holodex\viewmodel\state\UiState.kt
package com.example.holodex.viewmodel.state

/**
 * A generic class that represents a resource's state: Loading, Success, or Error.
 * This is used for individual shelves in the Discovery Hub and other async UI components.
 */
sealed class UiState<out T> {
    object Loading : UiState<Nothing>()
    data class Success<T>(val data: T) : UiState<T>()
    data class Error(val message: String) : UiState<Nothing>()
}

// File: res\drawable\ic_default_album_art_placeholder.xml
<vector xmlns:android="http://schemas.android.com/apk/res/android" android:height="24dp" andro
```

```
    <path android:fillColor="@android:color/white" android:pathData="M12,2C6.48,2 2,6.48 2,12s

</vector>


// File: res\drawable\ic_error_image.xml
<vector xmlns:android="http://schemas.android.com/apk/res/android" android:height="24dp" andro

    <path android:fillColor="@android:color/white" android:pathData="M11,15h2v2h-2zM11,7h2v6h-

</vector>


// File: res\drawable\ic_launcher_background.xml
<?xml version="1.0" encoding="utf-8"?>
<vector xmlns:android="http://schemas.android.com/apk/res/android"
    android:width="108dp"
    android:height="108dp"
    android:viewportWidth="108"
    android:viewportHeight="108">
    <path
        android:fillColor="#3DDC84"
        android:pathData="M0,0h108v108h-108z" />
    <path
        android:fillColor="#00000000"
        android:pathData="M9,0L9,108"
        android:strokeWidth="0.8"
        android:strokeColor="#33FFFFFF" />
    <path
        android:fillColor="#00000000"
        android:pathData="M19,0L19,108"
        android:strokeWidth="0.8"
        android:strokeColor="#33FFFFFF" />
    <path
        android:fillColor="#00000000"
        android:pathData="M29,0L29,108"
        android:strokeWidth="0.8"
        android:strokeColor="#33FFFFFF" />
    <path
        android:fillColor="#00000000"
        android:pathData="M39,0L39,108"
        android:strokeWidth="0.8"
        android:strokeColor="#33FFFFFF" />
    <path
        android:fillColor="#00000000"
        android:pathData="M49,0L49,108"
        android:strokeWidth="0.8"
        android:strokeColor="#33FFFFFF" />
    <path
        android:fillColor="#00000000"
        android:pathData="M59,0L59,108"
        android:strokeWidth="0.8"
        android:strokeColor="#33FFFFFF" />
    <path
```

```
    android:fillColor="#00000000"
    android:pathData="M69,0L69,108"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M79,0L79,108"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M89,0L89,108"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M99,0L99,108"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M0,9L108,9"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M0,19L108,19"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M0,29L108,29"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M0,39L108,39"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M0,49L108,49"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M0,59L108,59"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M0,69L108,69"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
```

```
    android:fillColor="#00000000"
    android:pathData="M0,79L108,79"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M0,89L108,89"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M0,99L108,99"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M19,29L89,29"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M19,39L89,39"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M19,49L89,49"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M19,59L89,59"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M19,69L89,69"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M19,79L89,79"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M29,19L29,89"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
    android:fillColor="#00000000"
    android:pathData="M39,19L39,89"
    android:strokeWidth="0.8"
    android:strokeColor="#33FFFFFF" />
<path
```

```
        android:fillColor="#00000000"
        android:pathData="M49,19L49,89"
        android:strokeWidth="0.8"
        android:strokeColor="#33FFFFFF" />
    <path
        android:fillColor="#00000000"
        android:pathData="M59,19L59,89"
        android:strokeWidth="0.8"
        android:strokeColor="#33FFFFFF" />
    <path
        android:fillColor="#00000000"
        android:pathData="M69,19L69,89"
        android:strokeWidth="0.8"
        android:strokeColor="#33FFFFFF" />
    <path
        android:fillColor="#00000000"
        android:pathData="M79,19L79,89"
        android:strokeWidth="0.8"
        android:strokeColor="#33FFFFFF" />
</vector>


// File: res\drawable\ic_launcher_foreground.xml
<vector xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:aapt="http://schemas.android.com/aapt"
    android:width="108dp"
    android:height="108dp"
    android:viewportWidth="108"
    android:viewportHeight="108">
    <path android:pathData="M31,63.928c0,0 6.4,-11 12.1,-13.1c7.2,-2.6 26,-1.4 26,-1.4l38.1,38
        <aapt:attr name="android:fillColor">
            <gradient
                android:endX="85.84757"
                android:endY="92.4963"
                android:startX="42.9492"
                android:startY="49.59793"
                android:type="linear">
                <item
                    android:color="#44000000"
                    android:offset="0.0" />
                <item
                    android:color="#00000000"
                    android:offset="1.0" />
            </gradient>
        </aapt:attr>
    </path>
    <path
        android:fillColor="#FFFFFF"
        android:fillType="nonZero"
        android:pathData="M65.3,45.828l3.8,-6.6c0.2,-0.4 0.1,-0.9 -0.3,-1.1c-0.4,-0.2 -0.9,-0.
        android:strokeWidth="1"
        android:strokeColor="#00000000" />
</vector>

// File: res\drawable\ic_like_empty.xml
```

```
<vector xmlns:android="http://schemas.android.com/apk/res/android" android:height="24dp" andro

    <path android:fillColor="@android:color/white" android:pathData="M480,480Q480,480 480,480Q

</vector>


// File: res\drawable\ic_notification_small.xml
<vector xmlns:android="http://schemas.android.com/apk/res/android" android:height="24dp" andro

    <path android:fillColor="@android:color/white" android:pathData="M12,3L4,9v12h16V9L12,3zM1

</vector>


// File: res\drawable\ic_pause.xml
<vector xmlns:android="http://schemas.android.com/apk/res/android" android:height="24dp" andro

    <path android:fillColor="@android:color/white" android:pathData="M12,2C6.48,2 2,6.48 2,12s

</vector>


// File: res\drawable\ic_placeholder_image.xml
<vector xmlns:android="http://schemas.android.com/apk/res/android" android:height="24dp" andro

    <path android:fillColor="@android:color/white" android:pathData="M20,2L8,2c-1.1,0 -2,0.9 -

</vector>


// File: res\drawable\ic_play_arrow.xml
<vector xmlns:android="http://schemas.android.com/apk/res/android" android:height="24dp" andro

    <path android:fillColor="@android:color/white" android:pathData="M8,5v14l11,-7z"/>

</vector>


// File: res\drawable\ic_repeat_off_24.xml
<vector xmlns:android="http://schemas.android.com/apk/res/android" android:height="24dp" andro

    <path android:fillColor="@android:color/white" android:pathData="M280,880L120,720L280,560L

</vector>


// File: res\drawable\ic_repeat_one_24.xml
<vector xmlns:android="http://schemas.android.com/apk/res/android" android:height="24dp" andro

    <path android:fillColor="@android:color/white" android:pathData="M7,7h10v3l4,-4 -4,-4v3L5,

</vector>
```

```
// File: res\drawable\ic_repeat_on_24.xml
<vector xmlns:android="http://schemas.android.com/apk/res/android" android:height="24dp" andro

    <path android:fillColor="@android:color/white" android:pathData="M120,920Q87,920 63.5,896.

</vector>


// File: res\drawable\ic_shuffle_off_24.xml
<vector xmlns:android="http://schemas.android.com/apk/res/android" android:height="24dp" andro

    <path android:fillColor="@android:color/white" android:pathData="M10.59,9.17L5.41,4 4,5.41

</vector>


// File: res\drawable\ic_shuffle_on_24.xml
<vector xmlns:android="http://schemas.android.com/apk/res/android" android:height="24dp" andro

    <path android:fillColor="@android:color/white" android:pathData="M120,920Q87,920 63.5,896.

</vector>


// File: res\drawable\ic_skip_next.xml
<vector xmlns:android="http://schemas.android.com/apk/res/android" android:height="24dp" andro

    <path android:fillColor="@android:color/white" android:pathData="M6,18l8.5,-6L6,6v12zM16,6

</vector>


// File: res\drawable\ic_skip_previous.xml
<vector xmlns:android="http://schemas.android.com/apk/res/android" android:height="24dp" andro

    <path android:fillColor="@android:color/white" android:pathData="M6,6h2v12L6,18zM9.5,12l8.

</vector>


// File: res\drawable\ic_stat_music_note.xml
<vector xmlns:android="http://schemas.android.com/apk/res/android" android:height="24dp" andro

    <path android:fillColor="@android:color/white" android:pathData="M12,3v10.55c-0.59,-0.34 -

</vector>


// File: res\drawable\ic_twitter.xml
<vector xmlns:android="http://schemas.android.com/apk/res/android"
    android:width="512dp"
    android:height="512dp"
    android:viewportWidth="512"
    android:viewportHeight="512">
    <path
```

```
        android:pathData="m0,0H512V512H0"
        android:fillColor="#fff"/>
    <path
        android:pathData="m458,140q-23,10 -45,12 25,-15 34,-43 -24,14 -50,19a79,79 0,0 0,-135
        android:fillColor="#1d9bf0"/>
</vector>

// File: res\drawable\ic_youtube.xml
<vector xmlns:android="http://schemas.android.com/apk/res/android"
    android:width="512dp"
    android:height="512dp"
    android:viewportWidth="512"
    android:viewportHeight="512">
    <path
        android:pathData="m0,0H512V512H0"
        android:fillColor="#fff"/>
    <path
        android:pathData="M313,256l-93,-53L220,309ZM427,169c9,37 9,138 0,174 -4,15 -17,27 -32,
        android:fillColor="#ed1d24"/>
</vector>

// File: res\drawable\twitter.xml
<vector xmlns:android="http://schemas.android.com/apk/res/android"
    android:width="24dp"
    android:height="24dp"
    android:viewportWidth="24"
    android:viewportHeight="24">
  <path
      android:pathData="M23,3a10.9,10.9 0,0 1,-3.14 1.53,4.48 4.48,0 0,0 -7.86,3v1A10.66,10.66
      android:strokeLineJoin="round"
      android:strokeWidth="2"
      android:fillColor="#00000000"
      android:strokeColor="#44819F"
      android:strokeLineCap="round"/>
</vector>


// File: res\drawable\youtube.xml
<vector xmlns:android="http://schemas.android.com/apk/res/android"
    android:width="24dp"
    android:height="24dp"
    android:viewportWidth="24"
    android:viewportHeight="24">
  <path
      android:pathData="M22.54,6.42a2.78,2.78 0,0 0,-1.94 -2C18.88,4 12,4 12,4s-6.88,0 -8.6,0.
      android:strokeLineJoin="round"
      android:strokeWidth="2"
      android:fillColor="#00000000"
      android:strokeColor="#44819F"
      android:strokeLineCap="round"/>
  <path
      android:pathData="M9.75,15.02l5.75,-3.27l-5.75,-3.27l0,6.54z"
      android:strokeLineJoin="round"
      android:strokeWidth="2"
      android:fillColor="#00000000"
```

```
      android:strokeColor="#44819F"
      android:strokeLineCap="round"/>
</vector>


// File: res\mipmap-anydpi-v26\ic_launcher.xml
<?xml version="1.0" encoding="utf-8"?>
<adaptive-icon xmlns:android="http://schemas.android.com/apk/res/android">
  <background android:drawable="@mipmap/ic_launcher_background"/>
  <foreground android:drawable="@mipmap/ic_launcher_foreground"/>
  <monochrome android:drawable="@mipmap/ic_launcher_monochrome"/>
</adaptive-icon>

// File: res\values\attrs.xml
<!-- res/values/attrs.xml -->
<resources>
    <attr name="notificationSmallIcon" format="reference"/>
</resources>

// File: res\values\colors.xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- Basic semantic colors, often overridden by themes -->
    <color name="black">#FF000000</color>
    <color name="white">#FFFFFFFF</color>

    <!-- Primary colors for XML themes (e.g., for Splash Screen or pre-Compose UI) -->
    <!-- These should ideally match or be inspired by your md_theme_light_primary -->
    <!-- from your Compose Color.kt for consistency if used by system UI elements -->
    <!-- before Compose takes over. -->
    <color name="seed">#6750A4</color> <!-- Your md_theme_light_primary -->

    <!-- Example: Colors matching your light theme (from your Color.kt) -->
    <!-- These can be used for things like windowSplashScreenBackground -->
    <color name="primary_light">#6750A4</color>
    <color name="on_primary_light">#FFFFFF</color>
    <color name="primary_container_light">#EADDFF</color>
    <color name="on_primary_container_light">#21005D</color>
    <color name="surface_light">#FFFBFE</color> <!-- Good for general backgrounds -->
    <color name="background_light">#FFFBFE</color>

    <!-- You might not need specific dark theme colors here if your -->
    <!-- DayNight theme handles it and Compose uses its own darkColorScheme. -->
    <!-- However, if you need to reference them from XML for some reason: -->
    <color name="primary_dark">#D0BCFF</color>
    <color name="on_primary_dark">#381E72</color>
    <color name="surface_dark">#1C1B1F</color> <!-- Good for general backgrounds in dark theme
    <color name="background_dark">#1C1B1F</color>

</resources>

// File: res\values\strings.xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- Existing strings -->
```

```
<string name="app_name">Holodex Music</string>
<string name="unknown_title">Unknown Title</string>
<string name="unknown_artist">Unknown Artist</string>
<string name="unknown_album">Unknown Album</string>
<string name="loading">Loading?</string>

<!-- API Key related -->
<string name="hint_api_key">Enter Holodex API Key</string>
<string name="button_save_key">Save Key</string>
<string name="toast_api_key_saved">API Key saved!</string>
<string name="toast_api_key_empty">API Key cannot be empty.</string>
<string name="toast_api_key_required">Please save an API Key first.</string>
<string name="status_api_key_required_fetch">Please enter and save your Holodex API Key to

<!-- Fetching & Data States -->
<string name="button_fetch_music">Fetch by Category</string>
<string name="button_load_more">Load More</string>
<string name="status_no_videos_found_filter">No videos found for this filter.</string>
<string name="status_select_category_first">Select a category first</string>
<string name="status_select_category_load_more">Fetch a category first to load more</strin
<string name="toast_fetching_songs_for">Fetching songs for: %s</string>
<string name="toast_playing_segment">Preparing to play segment: %s</string>
<string name="toast_playing_selected_segment">Preparing selected: %s</string>
<string name="toast_no_segments_playing_full">No segments. Preparing full audio for: %s</s
<string name="toast_stream_url_ready">Audio stream URL obtained! Ready for playback.</stri

<!-- Organization Spinner Prompt (Optional) -->
<string name="spinner_prompt_organization">Select Organization</string>
<string name="unknown_channel">Unknown Channel</string>
<string name="notification_no_media_title">No media playing</string>
<!-- Song List Dialog -->
<string name="toast_playing_segment_from_stream">Playing stream segment: %s</string>
<string name="unknown_song_title">Unknown Song Title</string>
<string name="song_timestamp_format">%1$s - %2$s</string>
<string name="timestamp_not_available">N/A</string>
<string name="dialog_title_select_song">Select Song</string>
<string name="button_play_full_stream">Play Full Stream</string>
<string name="cancel">Cancel</string>
<string name="unknown_song_title_short">Untitled Song</string>
<string name="song_count_format" translatable="false">%d songs</string>
<string name="more_options">More options</string>
<!-- Search Feature -->
<string name="hint_search_videos">Search music videos (e.g., song title, artist)</string>
<string name="button_search">Search</string>
<string name="status_enter_search_term">Please enter a search term.</string>
<string name="content_desc_channel_thumbnail">Channel Thumbnail</string>
<!-- Playback Notification / Media Session related -->
<string name="playback_notification_channel_name">Music Playback</string>
<string name="playback_notification_channel_description">Controls for music currently play
<!-- Potential error messages or states -->
<string name="error_stream_resolution_failed">Could not load stream.</string>
<string name="error_playback_failed">Playback error occurred.</string>
<string name="error_player_service_not_ready">Player service not ready. Please try again.<
<string name="error_playback_command_failed">Playback command failed (code: %d).</string>
<string name="error_initiating_playback">Error initiating playback.</string>
```

```
<!-- For MediaItem default descriptions or titles if primary ones are missing -->
<string name="default_media_item_title">Untitled Track</string>
<string name="default_media_item_artist">Unknown Artist</string>
<string name="action_previous">Previous</string>
<string name="action_play">Play</string>
<string name="action_pause">Pause</string>
<string name="action_next">Next</string>
<string name="action_like">Like</string>

<string name="action_back">Back</string>
<string name="nothing_selected_to_play">Nothing selected to play.</string>
<string name="action_shuffle">Shuffle</string>
<string name="action_repeat">Repeat</string>
<string name="action_view_queue">View Queue</string>

<string name="settings_title">Settings</string>
<string name="settings_section_api_key">API Key</string>
<string name="settings_section_cache">Cache Management</string>
<string name="settings_button_clear_cache">Clear All Application Cache</string>
<string name="settings_desc_clear_cache">This will remove all downloaded media, cached API
<string name="settings_section_about">About</string>
<string name="settings_app_version">App Version: %s</string>
<string name="control_panel_search_videos">Search Videos</string>
<string name="control_panel_browse_category">Browse by Category</string>

<string name="status_api_key_required_main">API Key is required to use the app. Please set
<string name="button_go_to_settings">Go to Settings</string>
<string name="status_no_videos_for_category">No videos found for the selected category.</s
<string name="category_favorites">Favorites</string>
<string name="action_unlike">Unlike</string>

<string name="action_clear_search">Clear search text</string>
<string name="search_history_title">Recent Searches</string>
<string name="action_clear_history">Clear History</string>
<string name="content_desc_search_history_item">Search history item</string>

<string name="settings_label_version">Version</string>
<string name="settings_label_powered_by">Powered By / Acknowledgements</string>
<string name="settings_link_holodex">Holodex.net</string>
<string name="settings_link_newpipe">NewPipeExtractor</string>
<string name="settings_section_theme">Appearance</string>
<string name="settings_theme_light">Light</string>
<string name="settings_theme_dark">Dark</string>
<string name="settings_theme_system">Follow System</string>
<string name="content_desc_external_link">Open external link</string>

<string name="dialog_title_create_playlist">Create New Playlist</string>
<string name="hint_playlist_name">Playlist Name</string>
<string name="hint_playlist_description_optional">Description (Optional)</string>
<string name="create">Create</string>

<string name="dialog_title_add_to_playlist">Add to playlist</string>
<string name="action_create_new_playlist_dialog">Create new playlist?</string>
<string name="action_more_options">More options</string>
<string name="action_add_to_playlist_menu">Add to playlist</string>
```

```
<string name="apply_filters_button">Apply Filters</string>
<string name="screen_title_playlist_details">Playlist Details</string>
<string name="message_playlist_is_empty">This playlist is empty.</string>
<string name="action_play_item">Play item</string>
<string name="action_remove_from_playlist">Remove from playlist</string>
<plurals name="item_count">
    <item quantity="one">%d item</item>
    <item quantity="other">%d items</item>
</plurals>
<string name="action_play_all">Play All</string>
<string name="action_play_all_playlist">Play all items in playlist</string>
<string name="message_no_favorited_videos">You haven\'t favorited any videos yet.</string>
<string name="message_no_liked_segments">You haven\'t liked any song segments yet.</string
<string name="action_play_all_liked_segments">Play All Liked Segments</string>
<string name="screen_title_playlists">My Playlists</string>
<string name="action_create_playlist">Create Playlist</string>
<string name="message_no_playlists_yet">No playlists yet. Tap the \'+\' button to create o
<string name="action_delete_playlist">Delete playlist</string>
<string name="category_display_name_search_results">Search: %s</string>
<string name="search_prompt_start">Search for music!</string>
<string name="playlist_title_liked_segments">Liked Segments</string>
<string name="playlist_desc_liked_segments">All your liked song segments</string>
<string name="message_no_liked_segments_in_playlist_view">You haven\'t liked any song segm
<string name="search_your_music_hint">Search your music</string>
<string name="action_menu">Menu</string>
<string name="action_filter_sort">Filter or Sort</string>
<string name="bottom_nav_browse">Browse</string>
<string name="bottom_nav_favorites">Favorites</string>
<string name="bottom_nav_playlists">Playlists</string>
<string name="bottom_nav_settings">Settings</string>
<string name="screen_title_favorites">My Favorites</string>
<string name="category_liked_segments">Liked Segments</string>
<string name="bottom_nav_search">Search</string>
<string name="screen_title_search">Search Music</string>
<string name="status_no_videos_for_filter">No videos found for the current filters.</strin

<!-- For FullPlayerScreen -->
<string name="content_desc_volume">Volume control</string>
<string name="action_hide_lyrics">Hide lyrics</string>
<string name="action_show_lyrics">Show lyrics</string>
<string name="action_show_player">Show player</string>
<string name="action_audio_settings">Audio settings</string>
<string name="action_view_artist">View artist</string>
<string name="action_view_album">View album</string>

<!-- General Player Strings -->
<string name="content_desc_album_art">Album artwork</string>
<string name="loading_track">Loading track?</string>
<string name="now_playing">Now Playing</string>
<string name="up_next_queue_title">Up Next</string>
<string name="empty_queue">Queue is empty</string>
<string name="action_clear_queue">Clear Queue</string>
<string name="action_remove_from_queue">Remove from queue</string>
<string name="currently_playing_indicator">Currently Playing</string>
<string name="action_move_up">Move up</string>
```

```
<string name="action_move_down">Move down</string>
<string name="content_desc_navigate_back">Navigate back</string>
<string name="content_desc_like_button">Like button</string>
<string name="content_desc_unlike_button">Unlike button</string>
<string name="content_desc_view_queue">View queue</string>
<string name="content_desc_more_options">More options</string>
<string name="content_desc_lyrics_toggle">Toggle lyrics</string>
<string name="content_desc_shuffle_button">Shuffle button</string>
<string name="content_desc_skip_previous_button">Skip to previous</string>
<string name="content_desc_play_pause_button">Play or Pause</string>
<string name="content_desc_skip_next_button">Skip to next</string>
<string name="content_desc_repeat_button">Repeat mode</string>
<string name="lyrics_not_available">Lyrics not available for this track.</string> <!-- ADD
<!-- For MainBrowseScreen & SearchScreen -->
<string name="status_search_no_results">No results found for \'%s\'.</string>
<string name="action_refresh">Refresh</string>
<string name="action_retry">Retry</string>
<string name="unknown_error_occurred">An unknown error occurred</string>
<string name="message_youve_reached_the_end">You\'ve reached the end!</string>
<string name="action_search">Search</string>
<string name="scroll_to_top">Scroll to top</string>
<string name="loading_content_message">Loading content?</string>
<string name="status_no_videos_for_filter_try_refresh">No music found for these filters. T
<string name="error_loading_generic">Could not load content: %s</string>
<string name="error_search_failed">Search failed: %s</string>
<string name="status_no_results_try_refresh">No results for \'%s\'. Try refreshing.</strin
<string name="error_playlist_not_found">Playlist not found.</string>

<string name="settings_section_data_performance">Data and Performance</string>
<string name="error_artist_info_unavailable">Artist/Channel information not available.</st
<string name="original_artist_label">(Original Artist)</string>
<string name="channel_label">(Channel)</string>
<string name="settings_label_image_quality">Image Quality</string>
<string name="settings_desc_image_quality">Lower quality reduces data usage and may speed
<string name="settings_image_quality_auto">Auto (Recommended)</string>
<string name="settings_image_quality_medium">Medium (Faster loading)</string>
<string name="settings_image_quality_low">Low (Data saver)</string>

<string name="settings_label_audio_quality">Audio Quality</string>
<string name="settings_desc_audio_quality">Select preferred audio quality for streaming. L
<string name="settings_audio_quality_best">Best Available</string>
<string name="settings_audio_quality_standard">Standard (~128kbps)</string>
<string name="settings_audio_quality_saver">Data Saver (~64kbps)</string>
<string name="action_add_to_queue_short">Add to Queue</string>
<string name="error_no_equalizer_app_found">No equalizer app found.</string>
<string name="error_no_audio_session">Audio session not available.</string>
<!-- NEW: Strings for Autoplay feature -->
<string name="settings_section_playback">Playback</string>
<string name="settings_label_autoplay_next_video">Autoplay next video</string>
<string name="settings_desc_autoplay_next_video">Automatically play the next song in the q
<string name="settings_label_data_loading">List Loading Intensity</string>
<string name="settings_desc_data_loading">"Normal" preloads more items for smoother scroll
<string name="settings_data_loading_normal">Normal (Smooth scrolling)</string>
<string name="settings_data_loading_reduced">Reduced (Less data usage)</string>
<string name="settings_label_list_loading_config">List Loading</string>
```

```
<string name="settings_desc_list_loading_config">"Adjust how aggressively lists load data.
<string name="settings_label_shuffle_on_play">Shuffle on Play</string>
<string name="settings_desc_shuffle_on_play">Automatically shuffle any new album, playlist
<string name="login_with_discord">Login with Discord</string>
<string name="settings_section_account">Account</string>
<string name="action_login">Login with Discord</string>
<string name="action_logout">Logout</string>
<string name="settings_desc_login">Log in to synchronize your history, likes, and playlist
<string name="content_desc_sync_icon">Sync status icon</string>

<!-- Display names for ListLoadingConfigOptions will be in SettingsScreen.kt enum -->

<string name="settings_label_buffering_strategy">Playback Buffering</string>
<string name="settings_desc_buffering_strategy">"Controls how much audio is buffered befor

<plurals name="queue_items_count">
    <item quantity="one">%d song</item>
    <item quantity="other">%d songs</item>
</plurals>

<string name="empty_queue_description">Add songs from the browse or search screens to get
<string name="drag_to_reorder">Drag to reorder</string>
<string name="remove_from_queue">Remove from queue</string>

<!-- For VideoDetailsScreen -->
<string name="video_thumbnail_description">Video thumbnail</string>
<string name="no_song_segments_available">No song segments are available for this video.</

<!-- Using plurals for song count for better localization -->
<plurals name="song_count">
    <item quantity="one">%d song</item>
    <item quantity="other">%d songs</item>
</plurals>
<string name="playlist_title_downloads">Downloads</string>
<string name="playlist_desc_downloads">All your downloaded songs</string>
<!-- For Toast message -->
<string name="added_songs_to_queue">Added %d songs to queue</string>
<string name="bottom_nav_home">Home</string>
<string name="bottom_nav_library">Library</string>
<string name="bottom_nav_downloads">Downloads</string>
<string name="message_no_favorites_or_segments">Your favorites and liked segments will app
<string name="action_download_all">Download All Segments</string>
<string name="search_results_title">Results for: %s</string>
<string name="action_clear_all">Clear</string>
<string name="settings_label_storage_location">Storage Location</string>
<string name="settings_desc_location_default">App-private storage (Default)</string>
<string name="settings_desc_location_custom">Custom Location: %s</string>
<string name="settings_button_reset_to_default">Reset to Default</string>
<string name="settings_label_download_location">Download Location</string>
<string name="settings_desc_download_location">Choose a folder for downloaded audio. Defau
<string name="settings_download_location_default">Default (App-private storage)</string>
<string name="action_clear_location">Clear custom location</string>
<string name="action_download">Download</string>
<string name="toast_download_started">Download started!</string>
<string name="status_download_failed">Download failed</string>
```

```
<!-- Titles for different download states in the notification -->
<plurals name="download_notification_title_in_progress">
    <item quantity="one">Downloading %d file</item>
    <item quantity="other">Downloading %d files</item>
</plurals>
<plurals name="download_notification_title_failed">
    <item quantity="one">%d download failed</item>
    <item quantity="other">%d downloads failed</item>
</plurals>
<string name="download_notification_title_completed">All downloads complete</string>
<string name="download_notification_text_failed">Could not download content. Tap to see de


<!-- Download Notification Channel details -->
<string name="download_notification_channel_name">Holodex Downloads</string>
<string name="download_notification_channel_description">Notifications for Holodex song do

<!-- Download Notification messages -->
<string name="download_notification_no_active_downloads">No active downloads</string>
<string name="download_notification_paused_reason">Downloads paused: %1$s</string>
<string name="download_notification_downloading_items_progress">Downloading %1$d items (%2
<string name="download_notification_processing_downloads">Processing downloads</string>

<!-- Download Requirement messages -->
<string name="download_requirement_network_connection">Network connection</string>
<string name="download_requirement_wifi_connection">Wi-Fi connection</string>
<string name="download_requirement_device_charging">Device charging</string>
<string name="download_requirement_battery_not_low">Battery not low</string>
<!-- Download Screen Strings -->
<string name="action_view_as_list">View as List</string>
<string name="action_view_as_grid">View as Grid</string>
<string name="search_your_downloads_hint">Search your downloads?</string>
<string name="action_cancel_download">Cancel download</string>
<string name="action_resume_download">Resume download</string>
<string name="action_retry_download">Retry download</string>
<string name="action_delete">Delete</string>
<string name="message_no_search_results_downloads">No downloads match your search</string>
<string name="message_no_downloads">No downloads yet\n\nStart downloading songs to see the

<!-- Download Status Strings -->
<string name="status_queued">Queued</string>
<string name="status_downloading">Downloading</string>
<string name="status_paused">Paused</string>
<string name="status_completed">Completed</string>
<string name="status_failed">Failed</string>
<string name="status_deleting">Deleting?</string>



<!-- Additional Action Strings -->
<string name="action_clear">Clear</string>
<string name="action_resume">Resume</string>
<string name="action_cancel">Cancel</string>
```

```
<!-- Progress and Status Messages -->
<string name="download_progress_percent">%1$d%%</string>
<string name="download_speed_format">%1$s/s</string>
<string name="download_eta_format">%1$s remaining</string>


<!-- Error Messages -->
<string name="error_download_failed">Download failed</string>
<string name="error_download_cancelled">Download cancelled</string>
<string name="error_network_unavailable">Network unavailable</string>
<string name="error_storage_full">Storage full</string>

<!-- Accessibility Strings -->
<string name="content_description_download_artwork">Artwork for %1$s</string>
<string name="content_description_download_progress">Download progress: %1$d percent</stri
<string name="content_description_download_status">Download status: %1$s</string>

<string name="screen_title_history">History</string>
<string name="message_no_history">Your listening history is empty.\n\nPlayed songs will ap
<string name="recently_played_songs">Recently Played Songs</string>
<string name="action_playAll">Play All</string>
<string name="action_add_to_queue">Add to Queue</string>
<plurals name="song_count_label">
    <item quantity="one">%d song</item>
    <item quantity="other">%d songs</item>
</plurals>
<string name="category_favorite_channels">Favorite Channels</string>

<string name="bottom_nav_discover">Discover</string>
<string name="action_show_more">Show More</string>
<string name="shelf_title_for_you">For You</string>
<string name="shelf_title_trending_songs">Trending Now</string>
<string name="shelf_title_recent_streams">Recent Live Performances</string>
<string name="shelf_title_fan_playlists">Community Mixes</string>
<string name="shelf_title_artist_radios">Artist Radios</string>
<string name="shelf_title_daily_mixes">Daily Mixes</string>
<string name="screen_title_more_results">More Results</string>
<string name="shelf_title_recent_streams_favorites">Recent streams from your favorites</st
<string name="content_desc_album_art_for">Album artwork for %1$s</string>

<string name="song_is_already_correctly_downloaded">\'%1$s\' is already correctly download
<string name="metadata_incorrect_reprocessing">Metadata incorrect, re-processing \'%1$s\'?
<string name="already_in_download_queue">\'%1$s\' is already in the download queue.</strin
<string name="retrying_export_for">Retrying export for \'%1$s\'?</string>
<string name="starting_download_for">Starting download for \'%1$s\'?</string>
<string name="verifying_songs_for_download">Verifying %1$d songs for download/repair?</str


<string name="sgp_artist_radio_title">%1$s Radio</string>
<string name="sgp_artist_radio_desc">Radio featuring %1$s</string>
<string name="sgp_daily_mix_title">Daily Mix: %1$s</string>
<string name="sgp_daily_mix_desc">Your daily dose of %1$s</string>
<string name="sgp_mv_random_title">Best of %1$s</string>
<string name="sgp_mv_latest_title">Recent %1$s Covers &amp;amp; Originals</string>
<string name="sgp_mv_random_desc">Relive the top hits from %1$s</string>
```

```
    <string name="sgp_mv_latest_desc">Latest released covers &amp;amp; originals from %1$s</string
    <string name="sgp_latest_title">Catch up on %1$s</string>
    <string name="sgp_latest_desc">Latest tagged songs in %1$s</string>
    <string name="sgp_weekly_mix_title">%1$s Weekly Mix</string>
    <string name="sgp_weekly_mix_desc">Explore this week in %1$s</string>
    <string name="sgp_my_weekly_mix_title">My Weekly Mix</string>
    <string name="sgp_my_weekly_mix_desc">Crafted for you based on your listening habits</stri
    <string name="sgp_history_title">Recently Played</string>
    <string name="sgp_history_desc">Your recently played songs</string>
    <string name="sgp_hot_title">Trending Songs</string>
    <string name="sgp_hot_desc">Trending songs radio</string>
    <string name="sgp_video_desc">Sang by %1$s</string>
    <string name="sgp_daily_mix_type">Daily Mix</string>
    <string name="sgp_weekly_mix_type">Weekly Mix</string>
    <string name="sgp_radio_type">Radio</string>
    <string name="action_share">Share</string>
    <string name="action_view_video">Go to Video</string>
</resources>

// File: res\values\Theme.xml
<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- Try this parent name -->
    <style name="Base.Theme.Holodex" parent="Theme.Material3.DayNight.NoActionBar">
        <!-- OR, if the above doesn't work, try the direct Material Components one (less likel
        <!-- <style name="Base.Theme.Holodex" parent="Theme.MaterialComponents.DayNight.NoActi
        <!-- Note: Theme.Material3.DayNight.NoActionBar IS the correct one for M3 -->

        <!-- Your items -->
        <item name="android:windowFullscreen">true</item>
        <item name="android:windowContentOverlay">@null</item>
        <item name="android:windowTranslucentStatus">false</item>
        <item name="android:windowTranslucentNavigation">false</item>
        <item name="android:fitsSystemWindows">false</item>
        <item name="android:windowLayoutInDisplayCutoutMode" tools:targetApi="p">shortEdges</i
    </style>

    <style name="Theme.Holodex" parent="Base.Theme.Holodex" />

    <!-- SplashScreen theme -->
    <style name="Theme.App.Starting" parent="Theme.SplashScreen">
        <item name="windowSplashScreenBackground">@color/primary_container_light</item>
        <item name="windowSplashScreenAnimatedIcon">@drawable/ic_launcher_foreground</item>
        <item name="postSplashScreenTheme">@style/Theme.Holodex</item>
    </style>
</resources>

// File: res\values-night\themes.xml
<resources xmlns:tools="http://schemas.android.com/tools">
    <style name="Base.Theme.Holodex" parent="Theme.Material3.DayNight.NoActionBar">
        <!-- Parent should be the same, DayNight handles the switch -->
        <!-- ... your dark theme specific attributes if any, usually handled by Compose ... --
    </style>
    <!-- No need to redefine Theme.Holodex here if it just inherits Base.Theme.Holodex -->
    <!-- If you had a Theme.App.Starting for night, it would go here too -->
</resources>
```

```
// File: res\xml\backup_rules.xml
<?xml version="1.0" encoding="utf-8"?><!--
   Sample backup rules file; uncomment and customize as necessary.
   See https://developer.android.com/guide/topics/data/autobackup
   for details.
   Note: This file is ignored for devices older than API 31
   See https://developer.android.com/about/versions/12/backup-restore
-->
<full-backup-content>
    <!--
   <include domain="sharedpref" path="."/>
   <exclude domain="sharedpref" path="device.xml"/>
-->
</full-backup-content>

// File: res\xml\data_extraction_rules.xml
<?xml version="1.0" encoding="utf-8"?><!--
   Sample data extraction rules file; uncomment and customize as necessary.
   See https://developer.android.com/about/versions/12/backup-restore#xml-changes
   for details.
-->
<data-extraction-rules>
    <cloud-backup>
        <!-- TODO: Use <include> and <exclude> to control what is backed up.
        <include .../>
        <exclude .../>
        -->
    </cloud-backup>
    <!--
    <device-transfer>
        <include .../>
        <exclude .../>
    </device-transfer>
    -->
</data-extraction-rules>
```