



GNSDK for Mobile (Android) Developers Guide

Release 1.1.9.4107

Published: 1/6/2016 11:22 AM

Gracenote, Inc.
2000 Powell Street, Suite 1500
Emeryville, California
94608-1804
www.gracenote.com

Preface

Preface	ii
Concepts	1
Introduction	1
<i>About Gracenote</i>	1
<i>What is Gracenote SDK?</i>	1
<i>Modules Overview</i>	1
Gracenote MetaData	6
<i>Gracenote Media Elements</i>	6
<i>Music Terminology</i>	6
<i>Genre and Other List-Dependent Values</i>	6
<i>Genres and List Hierarchies</i>	7
<i>Simplified and Detailed Hierarchical Groups</i>	8
<i>Core and Enriched Metadata</i>	8
<i>Mood and Tempo (Sonic Attributes)</i>	9
<i>Classical Music Metadata</i>	9
<i>Third-Party Identifiers and Preferred Partners</i>	11
Music Modules	12
<i>Music Module Overview</i>	12
<i>MusicID Overview</i>	12
CD TOC Recognition	14
TOC Identification	14
Multiple TOC Matches	14
Multiple TOCs and Fuzzy Matching	14
Text-Based Recognition	15
Fingerprint-Based Recognition	15
<i>About DSP</i>	15
<i>MusicID-File Overview</i>	15
Waveform and Metadata Recognition	16
Advanced Processing Methods	16

LibraryID	16
AlbumID	17
TrackID	17
MusicID-File Best Practices	17
Usage Notes	17
MusicID vs. MusicID-File	17
MusicID-Stream	18
Enriched Content Module (Link)	19
<i>Link (Content) Module Overview</i>	19
<i>Music Enrichment</i>	19
<i>Image Formats and Dimensions</i>	19
Available Image Dimensions	20
Common Media Image Dimensions	20
Music Cover Art	20
Artist Images	20
Genre Images	20
Discovery Features	21
<i>Playlists</i>	21
Collection Summaries	21
More Like This	21
.Playlist Requirements and Recommendations	21
Simplified Playlist Implementation	21
Playlist Content Requirements	21
Playlist Storage Recommendations	22
Playlist Resource Requirements	22
Playlist Level Equivalency for Hierarchical Attributes	22
Key Playlist Components	23
Media metadata: Metadata of the media items (or files)	23
Attributes: Characteristics of a media item, such as Mood or Tempo	23
Unique Identifiers	23
Collection Summary	24

Storage	24
Seed	24
Playlist generation	24
<i>Mood Overview</i>	24
Mood Descriptors	25
Mood Valence/Arousal Model	25
Mood Levels	25
Level 1 Valence/Arousal Map	26
Level 2 Valence/Arousal Map	26
Navigating with Mood	26
Slider Navigation	26
Grid Navigation	27
Bubble Magnitude Navigation	28
Other Mood Design Considerations	29
Mood Level Arousal/Valance Values	30
Level 1 Mood Levels	30
Level 2 Mood Levels	31
System Requirements	35
Supported Platforms and System Requirements	35
<i>Modules in the GNSDK Package</i>	35
GNSDK for Mobile Modules	35
<i>Memory Usage</i>	35
Memory Usage Guidelines	35
RAM Requirements	36
Playlist Memory Usage	37
Android	37
<i>Setting Up the Development Environment</i>	37
<i>Mobile Deployment</i>	38
Integrating into an Existing Android Project	38
GNSDK Android Permissions	38
<i>Advanced Implementations</i>	39

User History	39
Data Model	39
Sample Application User History Database Data Model	39
Adding Entries	40
Limiting Database Size	40
Recalling Entries	40
Mobile Getting Started	41
Getting Started with Mobile Android	41
Introduction	41
Fingerprint and Metadata Samples	41
Setting Up the Development Environment	41
Setting Up a Device	42
Creating the Sample Application	42
Music Recognition Widget	43
Adding Client ID and License Information to the Sample Application	43
Building and Running the Sample Application	43
Implementing Applications (All Platforms)	45
About the Implementing Applications Documentation	45
Basic Application Steps	45
Setup and Initialization	46
Authorizing a GNSDK Application	46
Client ID/Tag	47
License File	47
Including Header Files	47
Instantiating a GNSDK Manager Object	47
Specifying the License File	47
Instantiating a User Object	48
Saving the User Object to Persistent Storage	49
Loading a Locale	50
Locale-Dependent Data	51
Default Regions and Descriptors	52

<i>Locale Groups</i>	53
Multi-Threaded Access	53
Updating Locales and Lists	53
Update Notification	53
<i>Locale Behavior</i>	54
<i>Best Practices</i>	55
Using Gracenote SDK Logging	57
Enabling GNSDK Logging	57
Implementing Callback Logging	58
Enabling and Using GNSDK Storage	59
Enabling a Provider for GNSDK Storage	59
GNSDK Stores	60
GNSDK Databases	60
Setting GNSDK Storage Folder Locations	60
Getting Local Lookup Database Information	61
Image Information	61
Database Versions	61
Getting Available Locales	61
Setting Online Cache Expiration	62
Managing Online Cache Size and Memory	62
Using a Local Fingerprint Database	63
Downloading and Ingesting Bundles	63
Designating a Storage Provider	66
Setting Local and Online Lookup Modes	67
Supported Lookup Modes	67
Default Lookup Mode	68
Setting the Lookup Mode for a User or Queries	68
Using Both Local and Online Lookup Modes	69
Identifying Music	69
Identifying Music Using a CD-TOC, Text, or Fingerprints (MusicID)	69
MusicID Queries	69

Options for MusicID Queries	70
Identifying Music Using a CD TOC	70
Identifying Music Using Text	71
Identifying Music Using Fingerprints	72
PCM Audio Format Recommendations	72
MusicID Fingerprinting	73
Best Practices for MusicID Text Searches	75
<i>Identifying Audio Files (MusicID-File)</i>	77
Implementing an Events Delegate	77
Adding Audio Files for Identification	83
Setting Audio File Identification	84
MusicID-File Fingerprinting	85
Setting Options for MusicID-File Queries	88
Making a MusicID-File Query	89
Options When Making Query Call	89
<i>Identifying Streaming Music (MusicID-Stream)</i>	90
Setting Options for Streaming Audio Queries	93
<i>Accessing Classical Music Metadata</i>	93
Example Classical Music Output	93
Processing Returned Metadata Results	95
Needs Decision	96
Full and Partial Metadata Results	96
Locale-Dependent Data	97
Accessing Enriched Content	98
Setting the Query Option for Enriched Content	98
Processing Enriched Content	98
Retrieving a Content Asset	99
Retrieving and Parsing Enriched Content Code Samples:	99
Discovery Features	104
Generating a Playlist	104
Creating a Collection Summary	104

Populating a Collection Summary	105
Retrieving Playlist Attributes in Queries	106
How Playlist Gathers Data	106
Working with Local Storage	106
Generating a Playlist Using More Like This	107
Generating a Playlist Using PDL (Playlist Description Language)	108
Accessing Playlist Results	109
Working with Multiple Collection Summaries	112
Join Performance and Best Practices	112
Synchronizing Collection Summaries	113
Iterating the Physical Media	113
Processing the Collection	113
<i>Playlist PDL Specification</i>	113
PDL Syntax	113
Keywords	114
Example: Keyword Syntax	115
Operators	115
Literals	115
Attributes	116
Functions	116
PDL Statements	116
Attribute Implementation <att_imp>	117
Expression <expr>	117
Score <score>	117
Example: PDL Statements	117
<i>Implementing Mood</i>	118
Prerequisites	118
Enumerating Data Sources using Mood Providers	119
Creating and Populating a Mood Presentation	119
Iterating Through a Mood Presentation	120
Filtering Mood Results	122

Best Practices and Design Requirements	122
<i>Image Best Practices</i>	122
Image Resizing Guidelines	122
<i>UI Best Practices for Audio Stream Recognition</i>	123
Provide Clear and Accessible Instructions	124
Provide a Demo Animation	124
Display a Progress Indicator During Recognition	124
Use Animations During Recognition	124
Using Vibration, Tone, or Both to Indicate Recognition Complete	124
Display Help Messages for Failed Recognitions	124
Allow the User to Provide Feedback	124
Data Models	126
API Reference Documentation	127
Playlist PDL Specification	128
PDL Syntax	128
Keywords	128
Example: Keyword Syntax	129
Operators	130
Literals	130
Attributes	130
Functions	131
PDL Statements	131
Attribute Implementation <att_imp>	131
Expression <expr>	131
Score <score>	132
Example: PDL Statements	132
GNSDK Glossary	133
Index	139

Concepts

Introduction

About Gracenote

A pioneer in the digital media industry, Gracenote combines information, technology, services, and applications to create ingenious entertainment solutions for the global market.

From media management, enrichment, and discovery products to content identification technologies, Gracenote allows providers of digital media products and the content community to make their offerings more powerful and intuitive, enabling superior consumer experiences. Gracenote solutions integrate the broadest, deepest, and highest quality global metadata and enriched content with an infrastructure that services billions of searches a month from thousands of products used by hundreds of millions of consumers.

Gracenote customers include the biggest names in the consumer electronics, mobile, automotive, software, and Internet industries. The company's partners in the entertainment community include major music publishers and labels, prominent independents, and movie studios.

Gracenote technologies are used by leading online media services, such as Apple iTunes®, Pandora®, and Sony Music Unlimited, and by leading consumer electronics manufacturers, such as Pioneer, Philips, and Sony, and by nearly all OEMs and Tier 1s in the automotive space, such as GM, VW, Nissan, Toyota, Hyundai, Ford, BMW, Mercedes Benz, Panasonic, and Harman Becker.

For more information about Gracenote, please visit: www.gracenote.com.

What is Gracenote SDK?

Gracenote SDK (GNSDK) is a platform that delivers Gracenote technologies to devices, desktop applications, web sites, and backend servers. GNSDK enables easy integration of Gracenote technologies into customer applications and infrastructure—helping developers add critical value to digital media products, while retaining the flexibility to fulfill almost any customer need.

GNSDK is designed to meet the ever-growing demand for scalable and robust solutions that operate smoothly in high-traffic server and multithreaded environments. GNSDK is also lightweight—made to run in desktop applications and even to support popular portable devices.

Modules Overview

GNSDK consists of several modules that support specific Gracenote products. The principal module required for all applications is the GNSDK Manager. Other modules are optional, depending on the functionality of the applications you develop and the products you license from Gracenote.

The table below describes the primary modules provided by the GNSDK. The actual modules in your software package depends on your product and as specified in your Gracenote license agreement. The modules are listed alphabetically.

Manager module	Main module used by all applications, as it contains APIs that provide functionality common to the other GNSDK libraries. As Manager operates as the command center for all of the GNSDK libraries, the other GNSDK libraries cannot function without it. Any applications using GNSDK must have APIs implemented from the GNSDK libraries that meet their feature requirements, as well as APIs from Manager.
Link module	Provides links to enhanced and third-party music and video data. Examples include artist biographies, artist images, and more.
SQLite module	Provides a local storage solution for GNSDK using the SQLite database engine. This module is primarily used to manage a local cache of queries and content that the GNSDK libraries make to the Gracenote Service.
Submit module	Provides functionality necessary to submit metadata parcels to the Gracenote database.
MoodGrid module	Provides easy-to-use, spatially-aware music organization features to manage a user's music collection based on a set of Gracenote attributes. Built to provide a high-level user experience for local or cloud-based music collections and services that are Gracenote enabled, MoodGrid allows for single-point music categorization and playlist creation based on Mood. Additional filtering support is provided for GOET (Genre, Origin, Era, and Artist Type) and Tempo music attributes. MoodGrid: Generates a set of tracks based on a requested spatial mood index.
Music Fingerprinting (DSP) module	Provides digital signal processing functionality used by other GNSDK libraries.
Music Lookup Local module	Provides services for local lookup of various identification queries such as text and CD TOC search; plus provides services for local retrieval of content such as cover art. When enabled other GNSDK query objects are able to perform local lookups for various types of Gracenote searches.
Music Lookup Local Stream module	Provides services for local lookup of MusicID-Stream queries. The MusicID-Stream local database is constructed from "bundles" provided periodically by Gracenote. Your application must ingest the bundle, a process that adds the tracks from the bundle to the local database making them available for local recognition.

Music Lookup Local Stream2 module	Similar to Lookup Local Stream with an improved local fingerprint query engine to specify when to use the GPU for fingerprint queries instead of the CPU. Using the GPU may increase query performance.
MusicID module	Provides for audio recognition using CD TOC-based search (MusicID CD), Text-based searches (MusicID Text), Fingerprint, and Identifier lookup functionality. You must initialize the GNSDK DSP (digital signal processing) module before initializing the MusicID module, as the DSP module's functionality is necessary for MusicID's audio recognition functionality.
MusicID Match (Scan and Match) module	Provides music match capability to compare user music collections against a provider's music collection, such as a cloud-based service. Advanced music recognition fully integrates with customer catalog. Recognizes user's personal music collection syncs across devices. Upload only the unidentified tracks, dramatically reducing sync time. Only available through special arrangement with Gracenote.
MusicID Radio module	Supports automatically monitoring and recognizing radio audio streams. A subset of MusicID-Stream, it is designed to not require any end-user interaction. It automatically provides recognition and metadata retrieval of songs playing on the radio.
MusicID-File module	Provides a more comprehensive music file identification process than MusicID. MusicID-File is intended to be used with collections of file-based media. When an application provides decoded audio and text data for each file to the module, the MusicID-File functionality both identifies each file and groups the files into albums. Important: You must initialize the DSP (digital signal processing) module before initializing the MusicID-File module (DSP is required for audio recognition.)

MusicID-Stream module	Provides APIs to recognize music delivered as a continuous stream. It automatically manages buffering of streaming audio, and continuously fingerprints the audio stream. It does not retain these fingerprints for later re-use. If your application requires re-usable fingerprints, use MusicID or MusicID-File instead. After establishing an audio stream, the application can trigger an identification query at any time. For example, this action could be triggered on-demand by an end user pressing an "Identify" button provided by the application UI. The identification process identifies the buffered audio. Up to seven seconds of the most recent audio is buffered. If there is not enough audio buffered for identification, MusicID-Stream waits until enough audio is received. The identification process spawns a thread and completes asynchronously. The application can identify audio using Gracenote Service online database or a local database. The default behavior attempts a local database match. If this fails, the system attempts an online database match.
Playlist module	Supports creating lists of related songs from a music collection. It also supports "More Like This" playlists to generate optimal playlist results and eliminates the need to create and validate Playlist Definition Language statements.
Radio Station ID module	Supports looking up radio stations in a local database of radio station metadata and returns possible matches. Some examples of search criteria are call sign, broadcast frequency, market location, longitude and latitude of the car, which returns stations with nearby transmitters. Some of the returned Radio station metadata include: broadcast band, such as AM, FM, XM, HD; Branding, such as "California Public Radio"; Call sign; Broadcast frequency; Market location; and Logo.
Rhythm module	Supports creation of personalized adaptive radio stations and music recommendations for end users. Use seed artists and tracks, or a combination of genre, era, and mood to create the stations.
Taste module	Supports the creation of applications that can automatically understand an individual's listening preferences and habits by examining their relationship to music. Information concerning this relationship can come from various sources including their listening history, personal catalogs, social websites, music devices, or other sources of information. Taste analyzes this collection of songs and listening history and produces a persona. The persona is an abstraction of Gracenote attributes of all songs within it and is an accurate and compact representation of an individual's listening preferences and habits. The information stored within a persona can then be used to generate any number of channels. Each channel describes clusters of songs appropriate for the persona.

Video module	Supports all features related to looking up and working with video content and related metadata. Provides DVD and BluRay identification, and Text search for video; encompasses the VideoID and Video Explore products.
ACR module	Provides APIs used by Gracenote for Entourage for Automatic Content Recognition (ACR) of TV Programs
VACR module	Provides APIs used by Gracenote for Entourage for Automatic Content Recognition (VACR) of TV Programs
EPG module	Provides APIs used by Gracenote for Entourage for Electronic Programming Guide (EPG) features
FP Local module	Provides the option to store audio fingerprint data in local cache that the Entourage SDK can access for faster local lookup.

Gracenote MetaData

Gracenote Media Elements

Gracenote Media Elements are the software representations of real-world things like CDs, Albums, Tracks, Contributors, and so on. The following is a partial list of the higher-level media elements represented in GNSDK for Mobile:

Music

- Music CD
- Album
- Track
- Artist
- Contributor

Music Terminology

The following terms are used throughout the music documentation. For a detailed list of GNSDK terms and definitions, see the Glossary.

- **Album** - A collection of audio recordings, typically of songs or instrumentals.
- **Audio Work** - A collection of classical music recordings.
- **Track** - A song or instrumental recording.
- **Artist** - The person or group primarily responsible for creating the Album or Track.
- **Contributor** - A person that participated in the creation of the Album or Track.

Genre and Other List-Dependent Values

GNSDK for Mobile uses list structures to store strings and other information that do not directly appear in results returned from Gracenote Service. Lists generally contain information such as localized strings and region-specific information. Each list is contained in a corresponding List Type.

Some Gracenote metadata is grouped into hierarchical lists. Some of the more common examples of list-based metadata include genre, artist origin, artist era, and artist type. Other list-based metadata includes mood, tempo, roles, and others.

Lists-based values can vary depending on the locale being used for an application. That is, some values will be different depending on the chosen locale of the application. Therefore, these kinds of list-based metadata values are called *locale-dependent*.

Genres and List Hierarchies

One of the most commonly used kinds of metadata are genres. A genre is a categorization of a musical composition characterized by a particular style. The list system enables genre classification and hierarchical navigation of a music collection. The Genre list is very granular for two reasons:

- To ensure that music categories from different countries are represented and that albums from around the world can be properly classified and represented to users based on the user's geographic location and cultural preferences.
- To relate different regionally-specific music classifications to one another, to provide a consistent user experience in all parts of the world when creating playlists with songs that are similar to each other.

The Gracenote Genre System contains more than 2200 genres from around the world. To make this list easier to manage and give more display options for client applications, the Gracenote Genre System groups these genres into a relationship hierarchy. Most hierarchies consists of three levels: level-1, level-2, and level-3.

- Level-1
 - Level-2
 - Level-3

For example, the partial genre list below shows two, level-1 genres: Alternative & Punk and Rock.

Each of these genres has two, level-2 genres. For Rock, the level-2 genres shown are Heavy Metal and 50's Rock. Each level-2 genre has three level-3 genres. For 50's Rock, these are Doo Wop, Rockabilly, and Early Rock and Roll. This whole list represents 18 genres.

- Alternative & Punk
 - Alternative
 - Nu-Metal
 - Rap Metal
 - Alternative Rock
 - Punk
 - Classic U.K. Punk
 - Classic U.S. Punk
 - Other Classic Punk
- Rock
 - Heavy Metal
 - Grindcore
 - Black Metal
 - Death Metal
 - 50's Rock
 - Doo Wop
 - Rockabilly
 - Early Rock & Roll

Other category lists include: origin, era, artist type, tempo, and mood.

Simplified and Detailed Hierarchical Groups

In addition to hierarchical levels for some metadata, the Gracenote Genre System provides two general kinds of hierarchical groups (also called list hierarchies). These groups are called *Simplified* and *Detailed*.

You can choose which hierarchy group to use in your application for any of the locale/list-dependent values. Your choice depends on how granular you want the metadata to be.

The Simplified group retrieves the coarsest (largest) grain, and the Detailed group retrieves the finest (smallest) grain, as shown below.

Example of a Simplified Genre Hierarchy Group

- Level-1: 10 genres
 - Level-2: 75 genres
 - Level-3: 500 genres

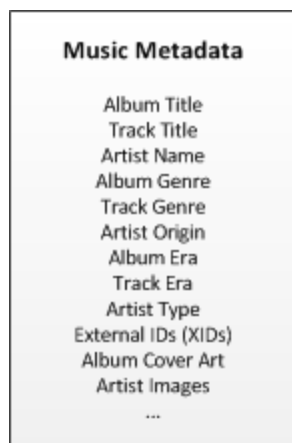
Example of a Detailed Genre Hierarchy Group

- Level-1: 25 genres
 - Level-2: 250 genres
 - Level-3: 800 genres

Core and Enriched Metadata

All Gracenote customers can access core metadata from Gracenote for the products they license. Optionally, customers can retrieve additional metadata, known as *enriched metadata*, by purchasing additional metadata entitlements.

The following diagram shows the core and enriched metadata available for Music.



Mood and Tempo (Sonic Attributes)

Gracenote provides two metadata fields that describe the sonic attributes of an audio track. These fields, mood and tempo, are track-level descriptors that capture the unique characteristics of a specific recording.

Mood is a perceptual descriptor of a piece of music, using emotional terminology that a typical listener might use to describe the audio track. Mood helps power Gracenote Playlist.

Tempo is a description of the overall perceived speed or pace of the music. Gracenote mood and tempo descriptor systems include hierarchical categories of increasing granularity, from very broad parent categories to more specific child categories.



NOTE: Tempo metadata is available online-only.

Classical Music Metadata

Gracenote also supports classical music metadata, which is typically more complex than non-classical music. Gracenote uses a Three-Line Solution (TLS) to map classical metadata to an AUDIO_WORK Track, Artist, and Album media elements. The tables below show this mapping.

Classical Music Three-Line Solution Metadata Mapping

Retrieved AUDIO_WORK Element	Returned Classical Music Metadata
Track	<p>[Composer Short Name]: [Work Title] In {Key}, {Opus}, {Cat#}, {"Nickname"} – [Movement#]. [M. Name]</p> <p>Example:</p> <p>Dvořák: Symphony #9 In E Minor, Op. 95, B 178, "From The New World" – 1. Adagio</p> <p>Dvořák [Composer Short Name]: Symphony #9 [Work Title] In E Minor {Key}, Op. 95 {Opus}, B 178 {Catalog #}, "From The New World" {Nickname} – 1. Adagio [Movement #] [M Name]</p>

Retrieved AUDIO_WORK Element	Returned Classical Music Metadata
Artist	<p>{Soloist(s)}, {Conductor}; {Ensemble #1}, {Ensemble #2, Etc.}</p> <p>Example:</p> <p>Anton Dermota, Cesare Siepi, Etc.; Josef Krips: Vienna Philharmonic Orchestra, Vienna State Opera Chorus</p> <p>Anton Dermota (Soloist), Cesare Siepi (Soloist), Etc. (indicates additional soloists); Josef Krips (Conductor): Vienna Philharmonic Orchestra (Ensemble #1), Vienna State Opera Chorus (Ensemble #2)</p>
Album	<p>Name printed on the spine of the physical CD, if available.</p> <p>Example: Music Of The Gothic Era [Disc 1]</p>

AUDIO_WORK Metadata Definitions

Album Name	<p>In most cases, a classical album's title is comprised of the composer(s) and work(s) that are featured on the product, which yields a single entity in the album name display. However, for albums that have formal titles (unlike composers and works), the title is listed as it is on the product.</p> <ul style="list-style-type: none"> • General title example: Beethoven: Violin Concerto • Formal title example: The Best Of Baroque Music
Artist Name	<p>A consistent format is used for listing a recording artist(s): by soloist(s), conductor, and ensemble(s), which yields a single entity in the artist name display.</p> <p>Example: Hilary Hahn; David Zinman: Baltimore Symphony Orchestra</p>
Catalog #	<p>The chronological number of a work as designated by a generally acknowledged scholar of a composer's work if it is a formal part of the Work Title. This is usually comprised of an upper-case letter and a number. The letter indicates either the last initial of the scholar/compiler of the composer's catalog. For Mozart, this would be "K" for Ludwig Ritter von Köchel) or the title of the scholar's catalog of the composer's works (for Bach, this would be "BWV" for "Bach-Werke-Verzeichnis (Bach Works List)."</p>
Composer	<p>The composer(s) that are featured on an album are listed by their last name in the album title (where applicable)</p>
Key	<p>The key signature of the work. It is included in a TLS data string if it is a formal part of the Work Title, such as Beethoven: Violin Concerto in D.</p> <p>If it is not part of the title (Debussy: Syrinx), it is not included in the data string. TLS exclusively uses the traditional key designations (A, B, C, D, etc.) as opposed to other versions (H-Moll, etc.)</p>

Movement	A constituent part or subsection of a work. A movement can be contained within a single track, or can comprise multiple tracks, but will always share the formal title of the parent work. Movements can, themselves, be subdivided into separate sections delineated by Act, Scene, Part, Variation, etc.
Movement #	The formally-sequenced number of a subsection of a work.
Nickname	This is an alternative title for a formal work that is used synonymously with that work. It is included in a TLS string if it is a formal part of the Work Title. For example: Beethoven: Symphony #9 In D Minor, Op. 125, "Choral"
Opus	Abbreviated as "Op." in the TLS string, this is the publishing number of the work in the composer's canon and is included in the data string if it is a formal part of the Work Title. For example: Strauss (R): Symphony In F Minor, Op. 12
Track	A distinct recorded performance, delineated by a specific time duration that can be quantified in a Table of Contents (TOC).
Track Name	A consistent format used for listing a track title—composer, work title, and (where applicable) movement title—which yields a single entity in the track name display. For example: Beethoven: Violin Concerto In D, Op. 61 – 1. Allegro Ma Non Troppo
Work Title	<p>For TLS purposes, the formal title of a musical work with the inclusion of the composer's Short Name prepended. Work titles come in two versions:</p> <ul style="list-style-type: none"> • "text" title comprising a word or set of words ("The Rite Of Spring"), or • title based on the musical form in which it was composed (Symphony #9, Sonata #1, etc.) <p>When applicable, a work title can also include work number, key signature, instrumental attribution, catalog number, opus number and/or nickname.</p> <p>Additional information can be included in parentheses, such as The Firebird (1911 Version), in the work title string. Modifiers such as Book, Suite, Vol., Part, etc. are also included in the title string.</p>

Third-Party Identifiers and Preferred Partners

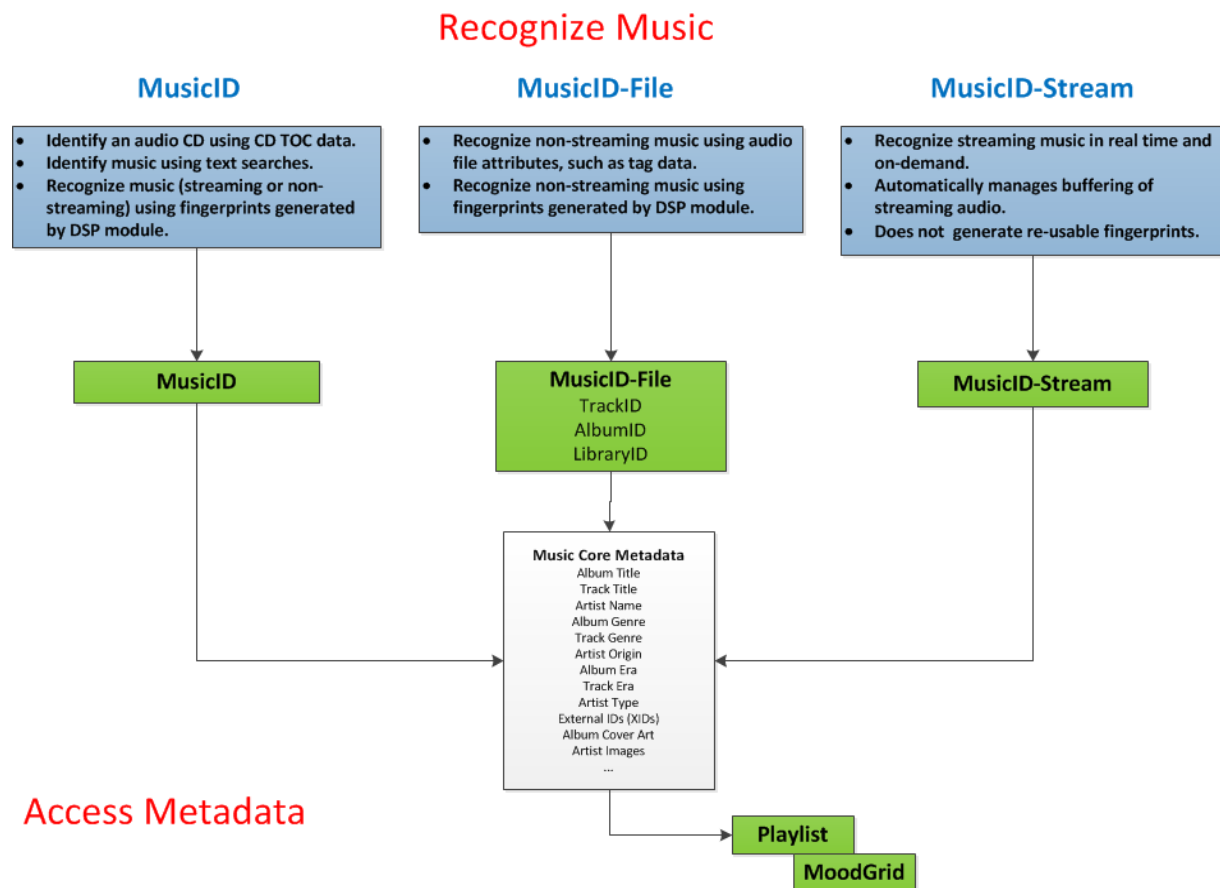
Link can match identified media with third-party identifiers. This allows applications to match media to IDs in stores and other online services—facilitating transactions by helping connect queries directly to commerce.

Gracenote has preferred partnerships with several partners and matches preferred partner content IDs to Gracenote media IDs. Entitled applications can retrieve IDs for preferred partners through Link.

Music Modules

Music Module Overview

The following diagram summarizes the kinds of identification queries each Music module supports.



MusicID Overview

MusicID allows application developers to deliver a compelling digital entertainment experience by giving users tools to manage and enjoy music collections on media devices, including desktop and mobile devices. MusicID is the most comprehensive identification solution in the industry with the ability to recognize, categorize and organize any music source, be it CDs, digital files, or audio streams. MusicID also seamlessly integrates with Gracenote's suite of products and provides the foundation for advanced services such as enriched content and linking to commerce.

Media recognition using MusicID makes it possible for applications to access a variety of rich data available from Gracenote. After media has been recognized, applications can request and utilize:

- Album, track, and artist names
- Genre, origin, era and type descriptors

GNSDK for Mobile accepts the following types of inputs for music recognition:

- CD TOCs
- File fingerprints
- Stream fingerprints
- Text input of album and track titles, album and track artist names, and composer names
- Media element identifiers
- Audio file and folder information (for advanced music recognition)

CD TOC Recognition

MusicID-CD is the component of GNSDK for Mobile that handles recognition of audio CDs and delivery of information including artist, title, and track names. The application provides GNSDK for Mobile with the TOC from an audio CD and MusicID-CD will identify the CD and provide album and track information.

TOC Identification

The only information that is guaranteed to be on every standard audio CD is a Table of Contents, or TOC. This is a header at the beginning of the disc giving the precise starting location of each track on the CD, so that CD players can locate the tracks and compute the track length information for their display panels.

This information is given in frames, where each frame is 1/75 of a second. Because this number is so precise, it is relatively unlikely that two unrelated CDs would have the same TOC. This lets Gracenote use the TOC as a relatively unique identifier.

The example below shows a typical TOC for a CD containing 13 tracks:

150 26670 52757 74145 95335 117690 144300 163992 188662 209375 231320 253150 281555 337792
--

The first 13 numbers represent the frames from the beginning of the disc that indicate the starting locations of the 13 tracks. The last number is the offset of the lead out, which marks the end of the CD program area.

Multiple TOC Matches

An album will often have numerous matching TOCs in the Gracenote database. This is because of CD manufacturing differences. More popular discs tend to have more TOCs. Gracenote maintains a catalog of multiple TOCs for many CDs, providing more reliable matching.

The following is an example of multiple TOCs for a single CD album. This particular album has 22 popular TOCs and many other less popular TOCs.

150 26670 52757 74145 95335 117690 144300 163992 188662 209375 231320 253150 281555 337642
--

150 26670 52757 74145 95335 117690 144300 163992 188662 209375 231320 253150 281555 337792
--

182 26702 52790 74177 95367 117722 144332 164035 188695 209407 231362 253182 281587 337675
--

150 26524 52466 73860 94904 117037 143501 162982 187496 208138 230023 251697 279880 335850
--

Multiple TOCs and Fuzzy Matching

Gracenote MusicID utilizes several methods to perform TOC matches. This combination of matching methods allows client applications to accurately recognize media in a variety of situations.

- Exact Match – when there is only one Product match for a queried CD TOC
- Multi-Exact Match – when there are multiple Product matches for a queried CD TOC
- Fuzzy Match – allows identification of media that has slight known and acceptable variations from well-recognized media.

Text-Based Recognition

You can identify music by using a lookup based on text strings. The text strings can be extracted from an audio track's file path name and from text data embedded within the file, such as mp3 tags. You can provide the following types of input strings:

1. Album title
2. Track title
3. Album artist
4. Track artist
5. Track composer

Text-based lookup attempts to match these attributes with known albums, artists, and composers. The text lookup first tries to match an album. If that is not possible, it next tries to match an artist. If that does not succeed, a composer match is tried. Adding as many input strings as possible to the lookup improves the results.

If a query handle populated with text inputs is passed to `gnsdk_musicid_find_matches()`, then best-fit objects will be returned. In this instance, you might get back album matches or contributor matches or both. Album matches are generally ranked higher than contributor matches

Fingerprint-Based Recognition

Gracenote uses audio fingerprinting as one method to identify tracks. Fingerprints can be generated from audio files and variety of audio sources, including recorded and degraded sources such as radios and televisions. This enables music identification using arbitrary audio sources—including sampling music via mobile devices.



Your application can retain fingerprints for a collection of audio files so they can be used later in queries. For example, your application can fingerprint an entire collection of files in a background thread and reuse them later.

About DSP

The DSP module is an internal module that provides Digital Signal Processing functionality used by other GNSDK for Mobile modules. This module is optional unless the application performs music identification or generates audio features for submission to Gracenote.

MusicID-File Overview

MusicID-File provides advanced file-based identification features not included in the MusicID module. MusicID-File can perform recognition using individual files or leverage collections of files to provide advanced recognition. When an application provides decoded audio and text data for each file to the library, MusicID-File identifies each file and, if requested, identifies groups of files as albums.

At a high level, MusicID-File APIs implement the following services:

- Identification through waveform fingerprinting and metadata
- Advanced processing methods for identifying individual tracks or file groupings and collections
- Result and status management

MusicID-File can be used with a local database, but it only performs text-matching locally. Fingerprints are not matched locally.



MusicID-File queries never return partial results. They always return full results.

Waveform and Metadata Recognition

The MusicID-File module utilizes both audio data and existing metadata from individual media files to produce the most accurate identification possible.

Your application needs to provide raw, decoded audio data (pulse-code modulated data) to MusicID-File, which processes it to retrieve a unique audio fingerprint. The application can also provide any metadata available for the media file, such as file tags, filename, and perhaps any application metadata. MusicID-File can use a combination of fingerprint and text lookups to determine a best-fit match for the given data.

The MusicID module also provides basic file-based media recognition using only audio fingerprints. The MusicID-File module is preferred for file-based media recognition, however, as its advanced recognition process provides significantly more accurate results.

Advanced Processing Methods

The MusicID-File module provides APIs that enable advanced music identification and organization. These APIs are grouped into the following three general categories - LibraryID, AlbumID, and TrackID.

LibraryID

LibraryID identifies the best album(s) for a large collection of tracks. It takes into account a number of factors, including metadata, location, and other submitted files when returning results. In addition, it automatically batches AlbumID calls to avoid overwhelming device and network resources.

Normal processing is 100-200 files at a time. In LibraryID, you can set the batch size to control how many files are processed at a time. The higher the size, the more memory will be used. The lower the size, the less memory will be used and the faster results will be returned. If the number of files in a batch exceeds batch size, it will attempt to make an intelligent decision about where to break based on other factors.

All processing in LibraryID is done through callbacks (for example, fingerprinting, setting metadata, returned statuses, returned results, and so on.). The status or result callbacks provide the only mechanism for accessing Response GDOs.



For object-oriented applications, for example, those written in C++, GDOs are the same thing as GnDataObjects. All object-oriented response objects are derived from GnDataObject.

AlbumID

AlbumID identifies the best album(s) for a group of tracks. For example, while the best match for a track would normally be the album where it originally appeared, if the submitted media files as a group are all tracks on a compilation album, then that is identified as the best match. All submitted files are viewed as a single group, regardless of location.

AlbumID assumes submitted tracks are related by a common artist or common album. Your application must be careful to only submit files it believes are related in this way. If your application cannot perform such grouping use LibraryID which performs such grouping internally

TrackID

TrackID identifies the best album(s) for a single track. It returns results for an individual track independent of any other tracks submitted for processing at the same time. Use TrackID if the original album a track appears on is the best or first result you want to see returned before any compilation, soundtrack, or greatest hits album the track also appears on.

MusicID-File Best Practices

- Use LibraryID for most applications. LibraryID is designed to identify a large number of audio files. It gathers the file metadata and then groups the files using tag data. LibraryID can only return a single, best match
- Use TrackID or AlbumID if you want all possible results for a track. You can request AlbumID and TrackID to return a single, best album match, or all possible matches. .
- Use AlbumID if your tracks are already pretty well organized by album. For memory and performance reasons, you should only provide a small number of related tracks. Your application should pre-group the audio files, and submit those groups one at a time.
- Use TrackID for one off track identifications and if the original album that a track appears on is the best or first result you want to see returned. TrackID is best for identifying outliers, that is those tracks unable to be grouped by the application for use with AlbumID. You can provide many files at once, but the memory consumed is directly proportional to the number of files provided. o Tkeep memory down you should submit a small number of files at a time

Usage Notes

- As stated above, TrackID and AlbumID are not designed for large sets of submitted files (more than an album's worth). Doing this could result in excessive memory use.
- For all three ID methods, you need to add files for processing manually, one-at-a-time. You cannot add all the files in a folder, volume, or drive in a single call.

MusicID vs. MusicID-File

Deciding whether to use the MusicID or MusicID-File SDK depends upon whether you are doing a "straightforward lookup" or "media recognition."

Use the MusicID SDK to perform a straightforward lookup. A lookup is considered straightforward if the application has a single type of data and would like to retrieve the Gracenote results for it. The source of the data does not matter (for example, the data might have been retrieved at a different time or from various sources). Examples of straightforward lookups are:

- Doing a lookup with text data only
- Doing a lookup with an audio fingerprint only
- Doing a lookup with a CD TOC
- Doing a lookup with a GDO value. Note that for object-oriented applications, for example, those written in C++, GDOs are the same thing as GnDataObjects. All object-oriented response objects are derived from GnDataObject.

Each of the queries above are completely independent of each other. The data doesn't have to come from actual media (for example, text data could come from user input). They are simply queries with a single, specific input.

Use the MusicID-File SDK to perform media recognition. MusicID-File performs recognition by using a combination of inputs. It assumes that the inputs are from actual media and uses this assumption to determine relationships between the input data. This SDK performs multiple straightforward lookups for a single piece of media and performs further heuristics on those results to arrive at more authoritative results. MusicID-File is capable of looking at other media being recognized to help identify results (for example, AlbumID).



If you only have a single piece of input, use MusicID. It is easier to use than MusicID-File, and for single inputs MusicID and MusicID-File will generate the same results.

MusicID-Stream

You can use MusicID-Stream to recognize music delivered as a continuous stream. Specifically, MusicID-Stream performs these functions:

- Recognizing streaming music in real time and on-demand .
- Automatically manages buffering of streaming audio.
- Continuously identifies the audio stream when initiated until it generates a response.

After establishing an audio stream, the application can trigger an identification query at any time. For example, this action could be triggered on-demand by an end user pressing an "Identify" button provided by the application UI.

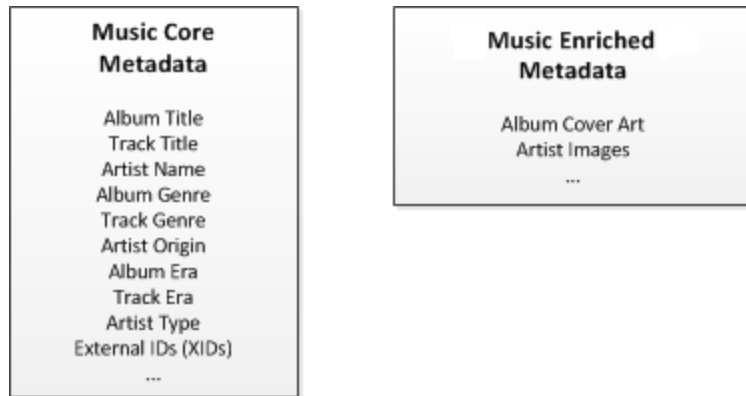
The identification process identifies the buffered audio. Up to seven seconds of the most recent audio is buffered. If there is not enough audio buffered for identification, MusicID-Stream waits until enough audio is received. The identification process spawns a thread and completes asynchronously.

The application can identify audio using Gracenote Service online database or a local database. The default behavior attempts a local database match. If this fails, the developer can attempt an online database match.

Enriched Content Module (Link)

Link (Content) Module Overview

The Link module allows applications to access *enriched content* beyond standard core metadata. The diagram below shows the difference between core metadata and enriched content for music.



Link allows applications to access and present enriched content related to media that has been identified using GNSDK identification features. Link delivers third-party content identifiers matched to the identified media, which can then be used to retrieve enriched content from Gracenote.

Enriched content offered through Gracenote Link includes:

- Music enrichment:
 - Album cover art
 - Artist images
 - Album reviews
 - Artist biographies

Music Enrichment

Link provides access to Gracenote Music Enrichment—a single-source solution for enriched content including cover art, artist images, biographies, and reviews. Link and Music Enrichment allow applications to offer enriched user experiences by providing high quality images and information to complement media.

Gracenote provides a large library of enriched content and is the only provider of fully licensed cover art, including a growing selection of international cover art. Music Enrichment cover art and artist images are provided by high quality sources that include all the major record labels.

Image Formats and Dimensions

Gracenote provides images in several dimensions to support a variety of applications. Applications or devices must specify image size when requesting an image from Gracenote. All Gracenote images are provided in the JPEG (.jpg) image format.

Available Image Dimensions

Gracenote provides images to fit within the following six square dimensions. All image sizes are available online and the most common 170x170 and 300x300 sizes are available in a local database.

Image Dimension Name	Pixel Dimensions
75	75 x 75
170	170 x 170
300	300 x 300
450	450 x 450
720	720 x 720
1080	1080 x 1080



Source images are not always square, and may be proportionally resized to fit within the specified square dimensions. Images will always retain their original aspect ratio.

Common Media Image Dimensions

Media images exist in a variety of dimensions and orientations. Gracenote resizes ingested images according to carefully developed guidelines to accommodate these image differences, while still optimizing for both developer integration and the end-user experience.

Music Cover Art

Although CD cover art is often represented by a square, it is commonly a bit wider than it is tall. The dimensions of these cover images vary from album to album. Some CD packages, such as a box set, might even be radically different in shape.

Artist Images

Artist and contributor images, such as publicity photos, come in a wide range of sizes and both portrait and landscape orientations.

Genre Images

Genre Images are provided by Gracenote to augment Cover Art and Artist Images when unavailable and to enhance the genre navigation experience. They are square photographic images and cover most of the Level 1 hierarchy items.

Discovery Features

Playlists

Playlist provides advanced playlist generation enabling a variety of intuitive music navigation methods. Using Playlist, applications can create sets of related media from larger collections—enabling valuable features such as More Like This™ and custom playlists—that help users easily find the music they want.

Playlist functionality can be applied to both local and online user collections. Playlist is designed for both performance and flexibility—utilizing lightweight data and extensible features.

Playlist builds on the advanced recognition technologies and rich metadata provided by Gracenote through GNSDK for Mobile to generate highly relevant playlists.

Collection Summaries

Collection summaries store attribute data to support all media in a candidate set and are the basis for playlist generation. Collection summaries are designed for minimal memory utilization and rapid consumption, making them easily applicable to local and server/cloud-based application environments.

Playlists are generated using the attributes stored in the active collection summary. Collection summaries must, therefore, be refreshed whenever media in the candidate set or attribute implementations are modified.

Playlist supports multiple collection summaries, enabling both single and multi-user applications.

More Like This

More Like This is a powerful and popular navigation feature made possible by Gracenote and GNSDK for Mobile Playlist. Using More Like This, applications can automatically create a playlist of music that is similar to user-supplied seed music. More Like This is commonly applied to an application's currently playing track to provide users with a quick and intuitive means of navigating through their music collection.

Gracenote recommends using More Like This to quickly implement a powerful music navigation solution. Functionality is provided via a dedicated API to further simplify integration. If you need to create custom playlists, you can use the Playlist Definition Language.

.Playlist Requirements and Recommendations

This topic discusses requirements and recommendations for your Playlist implementation.

Simplified Playlist Implementation

Gracenote recommends streamlining your implementation by using the provided More Like This function, `gnsdk_playlist_generate_morelikethis()`. It uses the More Like This algorithm to generate optimal playlist results and eliminates the need to create and validate Playlist Definition Language statements.

Playlist Content Requirements

Implementing Playlist has these general requirements:

- The application integrates with GNSDK for Mobile's MusicID or MusicID-File (or both) to recognize music media and create valid GDOs. Note that for object-oriented applications, for example, those written in C++, GDOs are the same thing as GnDataObjects. All object-oriented response objects are derived from GnDataObject.
- The application uses valid unique identifiers. A unique identifier must be a valid UTF-8 string of unique media identifier data. For more information, see "Unique Identifiers" on page 23.



Unique identifiers are essential to the Playlist generation process. The functionality cannot reference a media item if its identifier is missing or invalid.

Playlist Storage Recommendations

GNSDK for Mobile provides the SQLite module for applications that may need a storage solution for collections. You can dynamically create a collection and release it when you are finished with it. If you choose this solution, you must store the GDOs or recognize the music at the time of creating the collection.

Your application can also store the collection using the serialization and deserialization functions.

Playlist Resource Requirements

The following table lists resource requirements for Playlist's two implementation scenarios:

Use Case	Typical Scenario	Number of Collection Summaries	Application Provides Collection Summary to Playlist	Required Computing Resources
Single user	Desktop user Mobile device user	Generally only one	Once, normally at start-up	Minimal-to-average, especially as data is ingested only once or infrequently
Multiple users	Playlist server Playlist - in-the-cloud system	Multiple; requires a unique collection summary for each user who can access the system	Dynamically and multiple times; typically loaded with the playlist criteria at the moment before playlist generation	Requires more computing resources to ensure an optimal user experience

Playlist Level Equivalency for Hierarchical Attributes

Gracenote maintains certain attribute descriptors, such as Genre, Era, Mood, and Tempo, in multi-level hierarchies. For a descriptions of the hierachies, see "Mood and Tempo (Sonic Attributes)" on page 9. As such, Playlist performs certain behaviors when evaluating tracks using hierarchical attribute criteria.

Track attributes are typically evaluated at their equivalent hierarchy list-level. For example, Rock is a Level 1 genre. When evaluating candidate tracks for a similar genre, Playlist analyzes a track's Level 1 genre.

However, Seeds contain the most granular-level attribute. When using a SEED, Playlist analyzes tracks at the respective equivalent level as is contained in the Seed, either Level 2 or Level 3.

Key Playlist Components

Playlist operates on several key components. The GNSDK for Mobile Playlist module provides functions to implement and manage the following key components within your application.

Media metadata: Metadata of the media items (or files)

The media may be on MP3s on a device, or a virtual collection hosted on a server. Each media item must have a unique identifier, which is application-defined.

Playlist requires recognition results from GNSDK for Mobile for operation, and consequently must be implemented with one or both of GNSDK for Mobile's music identification modules, MusicID and MusicID-File.

Attributes: Characteristics of a media item, such as Mood or Tempo

Attributes are Gracenote-delivered string data that an application can display; for example, the Mood attribute Soulful Blues.

When doing recognition queries, if the results will be used with Playlist, set the 'enable playlist' query option to ensure proper data is retrieved for the result (GNSDK_MUSICID_OPTION_ENABLE_PLAYLIST or GNSDK_MUSICIDFILE_OPTION_ENABLE_PLAYLIST).

Unique Identifiers

When adding media to a collection summary, an application provides a unique identifier and a GDO which contains the metadata for the identifier.



For object-oriented applications, for example, those written in C++, GDOs are the same thing as GnDataObjects. All object-oriented response objects are derived from GnDataObject.

The identifier is a value that allows the application to identify the physical media being referenced. The identifier is not interpreted by Playlist; it is only returned to the application in Playlist results. An identifier is generally application-dependent. For example, a desktop application typically uses a full path to a file name for an identifier, while an online application typically uses a database key. The media GDO should contain relevant metadata for the media referenced by the identifier. In most cases the GDO comes from a recognition event for the respective media (such as from MusicID). Playlist will take whatever metadata is relevant for playlist generation from the given GDO. For best results, Gracenote recommends giving Album Type GDOs that have matched tracks to this function; Track Type GDOs also work well. Other GDOs are supported, but most other types lack information for good Playlist generation.

Collection Summary

A collection summary contains the distilled information GNSDK for Mobile Playlist uses to generate playlists for a set of media.

Collection summaries must be created and populated before Playlist can use them for playlist generation. Populating collection summaries involves passing a GDO from another GNSDK for Mobile identification event (for example, from MusicID) along with a unique identifier to Playlist. Collection Summaries can be stored so they do not need to be reconstructed before every use.

Storage

The application can store and manage collection summaries in local storage.

Seed

A seed is the GDO of a media item in the collection summary used as input criteria for playlist generation. A seed is used to generate More Like This results, or in custom PDL statements. For example, the seed could be the GDO of a particular track that you'd like to use to generate More Like This results.

Playlist generation

GNSDK for Mobile provides two Playlist generation functions: a general function for generating any playlist, `gnsdk_playlist_generate_playlist()`, and a specific function for generating a playlist that uses the Gracenote More Like This algorithm, `gnsdk_playlist_generate_morelikethis()`.



NOTE: Gracenote recommends streamlining your Playlist implementation by using the provided More Like This™ function, `gnsdk_playlist_generate_morelikethis()`, which uses the More Like This algorithm to generate optimal playlist results and eliminates the need to create and validate PDL statements.

Mood Overview

Gracenote provides track-based mood data that allow users to generate playlists based on the mood they want to listen to. When the user selects a mood, the application provides a playlist of music that corresponds to the selected mood.

An important advantage of Mood is that it is track-specific and independent of other track metadata. This enables you to create intuitive user interfaces that can create mood-based playlists across different genres, eras, or artist types, and so on. The user can also filter the playlist using one or more of these other attributes to refine the playlist.

For details on implementing Mood features, see [Implementing Mood](#)

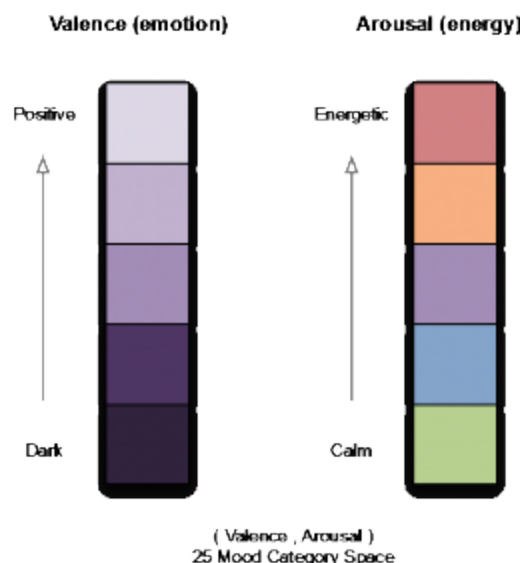
Mood Descriptors

MusicID, MusicID-File, and MusicID-Stream return mood metadata in track results. Gracenote defines over 100 mood types. This granularity of moods is useful in non-automotive applications, such as desktop and tablet interfaces. To simplify mood selection for with limited size interfaces, Gracenote provides a superset of 25 moods called Level 1. Examples of Level 1 range from Peaceful to Excited and from Somber to Aggressive.

Mood Valence/Arousal Model

Gracenote characterizes moods using a Valence/Arousal model. The mood of every track in the Gracenote repository is expressed as a coupled value of (Valence/Arousal):

- Valence (Emotion) is a psychological term for defining the positivity or negativity of emotions. Valence describes the attractiveness (positive valence) or averseness (negative valence) of an event, object, or situation. For example, the emotions popularly referred to as negative, such as anger and fear, have negative valence. Emotions popularly referred to as positive, such as joy and peacefulness, have positive valence.
- Arousal (Energy) is a psychological term for describing the energy associated with an emotion. For example, the emotions associated with peaceful are considered to have low arousal, while the emotions associated with celebratory have a high arousal.



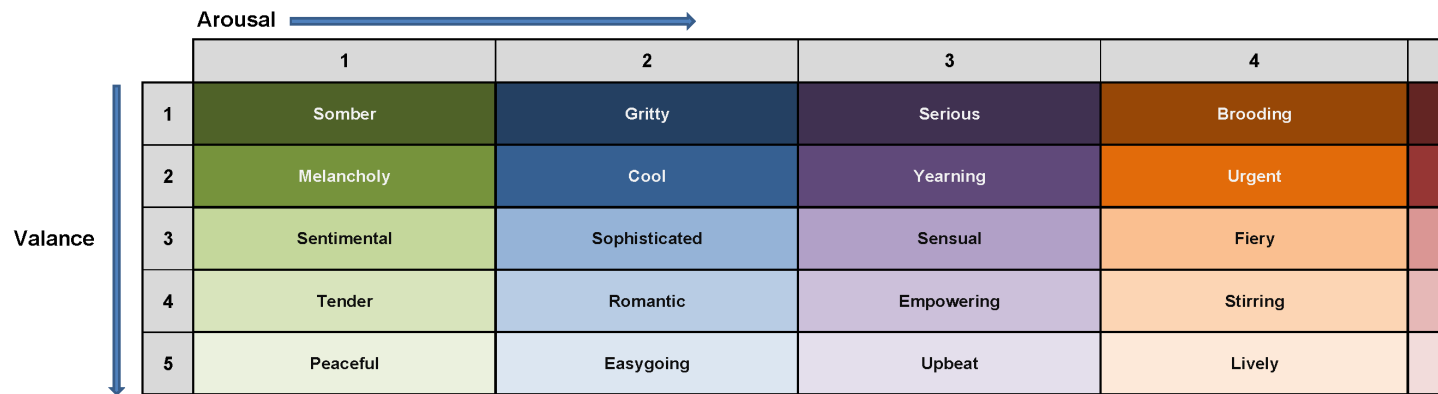
Mood Levels

The following tables list mood levels for Level 1 and Level 2 categories. In these examples, the top row of numbers represents Arousal (Energy) values, and the column of numbers on the left represent the Valence (Emotion). For example in Level 1: Peaceful is (1,5), indicating a low Arousal of 1, and high Valence of 5. On the contrary, Aggressive is (5,1) indicating high Arousal of 5, and a low Valence of 1.

For a list of Valence/Arousal value mappings for each mood level, see [Mood Level Arousal/Valence Values](#).

Level 1 Valence/Arousal Map

L1 (25) Category Sonic Mood Grid



Level 2 Valence/Arousal Map

L2 (100) Category Sonic Mood Grid



Navigating with Mood

This topic suggests some possible UI designs for mood-based playlists. The designs presented are suggestions only. The Mood APIs are flexible and can support most any type of UI that can be designed.

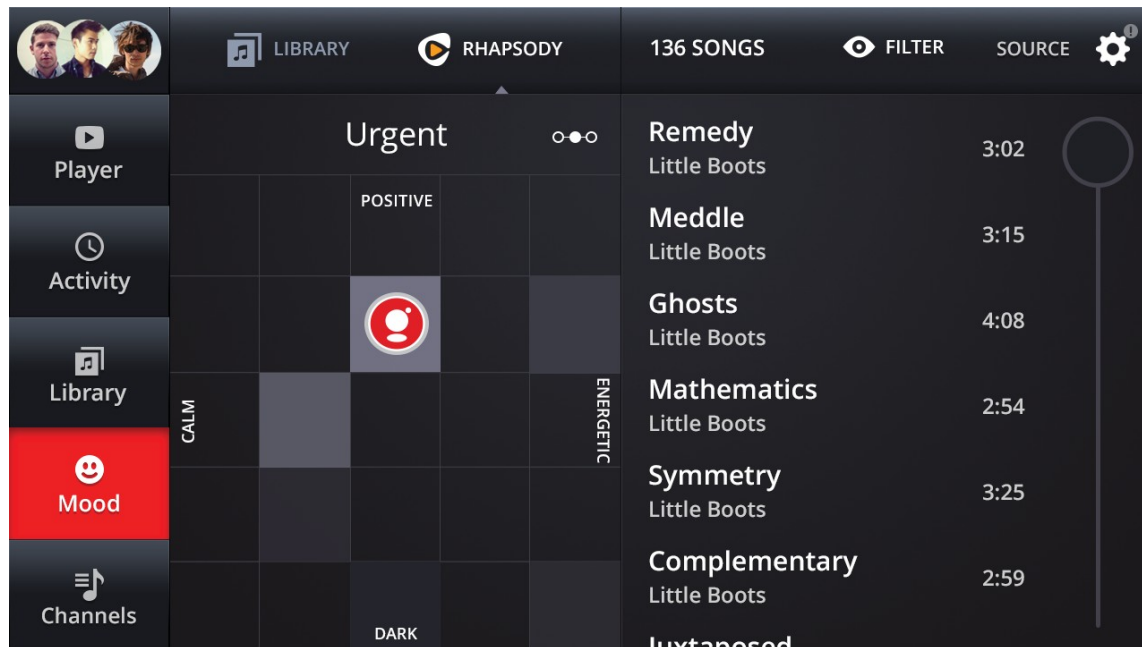
Slider Navigation

Slider Mood Navigation is shown below. Gracenote recommends this design because of its simplicity and ease-of-use in an automotive environment. It can support a touch-screen or scroll-wheel interface for selecting values. This design provides the ability to choose five discrete values on each dimension of Valence and Arousal.



Grid Navigation

Grid navigation is shown below. It is similar to the slider design. However, mood selection is made across a two-dimensional grid of discrete mood values. With this design, the user can select 25 discrete values as cells on a two-dimensional grid of Valence/Arousal values. This design supports a touch-screen easily, but may be more difficult to map to a scroll-wheel interface. Grid navigation can implement a heat-map design to indicate the relative number of tracks matching each grid. For example, the more tracks there are in a cell, the darker the shade of the cell. In this example, the vertical axis corresponds to Valence values, and the horizontal axis corresponds to Arousal values.

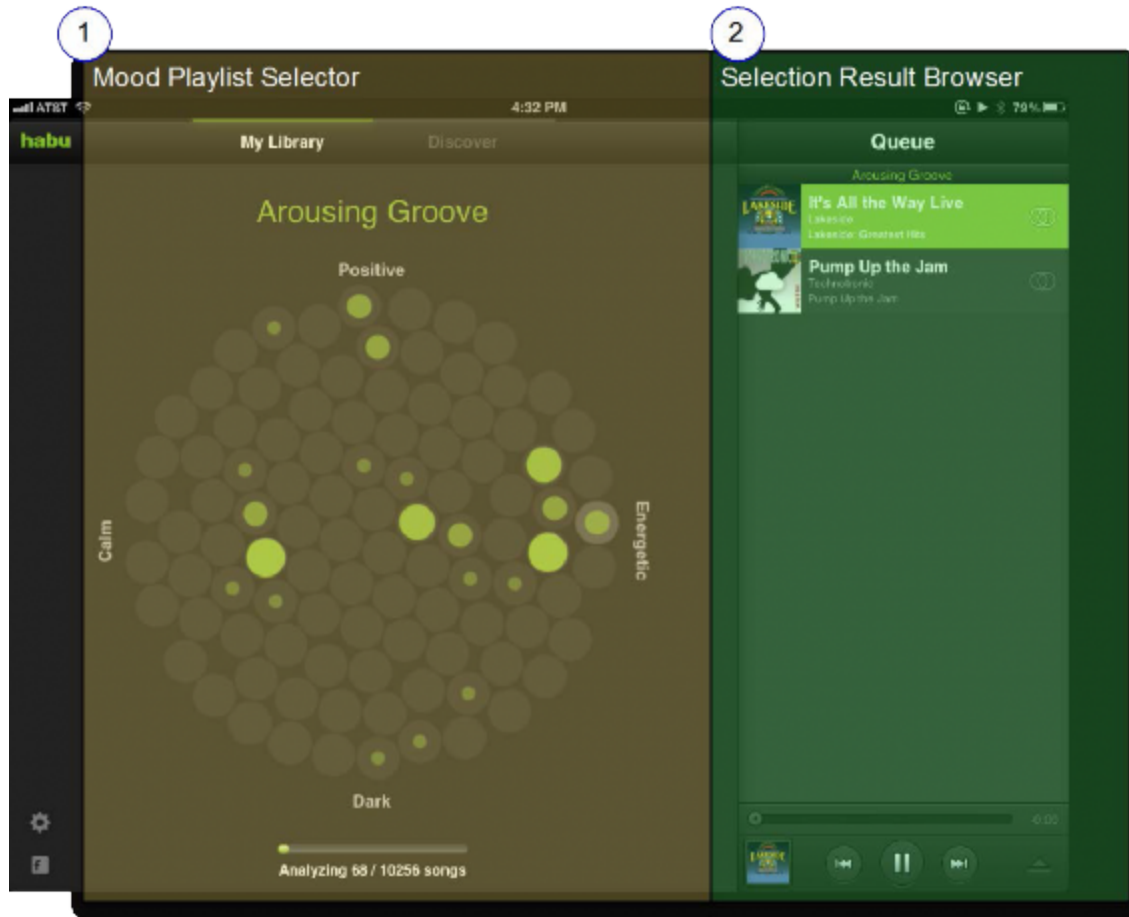


Bubble Magnitude Navigation

Bubble Magnitude navigation is shown below. This interface differs by using state bubbles to show the user what moods exist in the collection. Bubbles that are filled-in contain tracks for that mood. The size of the bubble indicates the relative number of tracks (the magnitude) in that mood category. This is similar to the heat-map design used in Grid navigation.

The bubbles are placed in a Valence/Arousal grid to provide an intuitive way to select mood playlists. The implementation supports a touch screen easily, but may be more difficult to map to a scroll-wheel interface..

The example below shows Bubble Magnitude navigation for the Level 2 set of 100 moods. This implementation is based on the Gracenote Habu mobile application for the iPad. Automotive applications should provide Level 1 moods (25 bubbles). As in the other implementations, selecting a bubble creates a playlist for the Valence/Arousal value.



Other Mood Design Considerations

Whichever interface design you implement, your application should follow these guidelines when a user selects a mood.

- Keep the Mood Playlist Selector active so users can easily select different moods.
- When the user selects a new mood, go directly to the Selection Results Browser with the refreshed playlist.
- After several seconds of Mood being active, go directly to Selection Results Browser and start playback of the tracks. This allows the user to activate Mood and continue to use the mood selection from the last time Mood was used.

In addition to the UI designs presented above, you can add other components to enhance the mood experience. Examples of these enhancements are:

- Voice Commands: Mood playlist navigation by voice is a feature that can improve the automobile music listening experience. As a mood playlist is playing for instance, a user could speak More Positive or Less Dark to have the playlist automatically adjust to their tastes. All of the interfaces

presented above could support a voice interface like this, but the Slider navigation interface visually maps more directly to these commands.

- **Genre Filtering:** The Grid navigation interface can be further enhanced to include a genre filter button. This filter lets the user to refine mood playlist to specific genres. To keep the filter selection simple, limit the available genres to only those in the user's collection. Your application can provide filters based on other track metadata as well, such as era and artist type.
- **100 Mood Option:** Both Grid and Bubble Magnitude navigation interfaces can support Level 2 (100 moods). The Slider navigation interface can also support Level 2 using two dimensions of 10 buttons each. However, the Level 1 (25 mood) design is optimal for this interface.

Mood Level Arousal/Valance Values

The tables below list the Valence/Arousal value mappings for Level 1 and Level 2 mood levels.

Level 1 Mood Levels

Mood	Valance (Emotion)	Arousal (Energy)
Somber	1	1
Melancholy	2	1
Sentimental	3	1
Tender	4	1
Peaceful	5	1
Gritty	1	2
Cool	2	2
Sophisticated	3	2
Romantic	4	2
Easygoing	5	2
Serious	1	3
Yearning	2	3
Sensual	3	3
Empowering	4	3
Upbeat	5	3
Brooding	1	4
Urgent	2	4

Mood	Valance (Emotion)	Arousal (Energy)
Fiery	3	4
Stirring	4	4
Lively	5	4
Aggressive	1	5
Defiant	2	5
Energizing	3	5
Rowdy	4	5
Excited	5	5

Level 2 Mood Levels

Mood	Valance (Emotion)	Arousal (Energy)
Dark Cosmic	1	1
Solemn / Spiritual	2	1
Wistful / Forlorn	3	1
Mysterious / Dreamy	4	1
Lyrical Sentimental	5	1
Tender / Sincere	6	1
Romantic / Lyrical	7	1
Refined / Mannered	8	1
Reverent / Healing	9	1
Pastoral / Serene	10	1
Creepy / Ominous	1	2
Enigmatic / Mysterious	2	2
Sad / Soulful	3	2
Light Melancholy	4	2
Cool Melancholy	5	2
Gentle Bittersweet	6	2
Light Groovy	7	2

Mood	Valance (Emotion)	Arousal (Energy)
Awakening / Stately	8	2
Quiet / Introspective	9	2
Delicate / Tranquil	10	2
Depressed / Lonely	1	3
Sober / Determined	2	3
Cool Confidence	3	3
Casual Groove	4	3
Intimate Bittersweet	5	3
Suave / Sultry	6	3
Dramatic / Romantic	7	3
Sweet / Sincere	8	3
Friendly	9	3
Hopeful / Breezy	10	3
Gritty / Soulful	1	4
Strumming Yearning	2	4
Dark Groovy	3	4
Wary / Defiant	4	4
Smoky / Romantic	5	4
Dark Playful	6	4
Lush / Romantic	7	4
Heartfelt Passion	8	4
Charming / Easygoing	9	4
Cheerful / Playful	10	4
Serious / Cerebral	1	5
Melodramatic	2	5
Sensitive / Exploring	3	5
Bittersweet Pop	4	5
Dreamy Pulse	5	5

Mood	Valance (Emotion)	Arousal (Energy)
Soft Soulful	6	5
Dramatic Emotion	7	5
Strong / Stable	8	5
Soulful / Easygoing	9	5
Carefree Pop	10	5
Thrilling	1	6
Hypnotic Rhythm	2	6
Energetic Dreamy	3	6
Energetic Yearning	4	6
Intimate	5	6
Sensual Groove	6	6
Idealistic / Stirring	7	6
Powerful / Heroic	8	6
Happy / Soulful	9	6
Party / Fun	10	6
Dreamy Brooding	1	7
Evocative / Intriguing	2	7
Dark Urgent	3	7
Dark Pop	4	7
Passionate Rhythm	5	7
Dark Sparkling Lyrical	6	7
Focused Sparkling	7	7
Invigorating / Joyous	8	7
Playful / Swingin	9	7
Showy / Rousing	10	7
Alienated / Brooding	1	8
Energetic Melancholy	2	8
Energetic Anxious	3	8

Mood	Valance (Emotion)	Arousal (Energy)
Dark Pop Intensity	4	8
Energetic Abstract Groove	5	8
Fiery Groove	6	8
Triumphant / Rousing	7	8
Jubilant / Soulful	8	8
Exuberant / Festive	9	8
Lusty / Jaunty	10	8
Chaotic / Intense	1	9
Dark Hard Beat	2	9
Attitude / Defiant	3	9
Heavy Brooding	4	9
Edgy / Sexy	5	9
Arousing Groove	6	9
Confident / Tough	7	9
Ramshackle / Rollicking	8	9
Upbeat Pop Groove	9	9
Loud Celebratory	10	9
Aggressive Power	1	10
Heavy Triumphant	2	10
Hard Dark Excitement	3	10
Hard Positive Excitement	4	10
Abstract Beat	5	10
Heavy Beat	6	10
Driving Dark Groove	7	10
Wild / Rowdy	8	10
Happy Excitement	9	10
Euphoric Energy	10	10

System Requirements

Supported Platforms and System Requirements

GNSDK for Mobile provides multi-threaded, thread-safe technology for a wide variety of platforms and languages. For a detailed list of these, see the Release Notes included in the software package.

Modules in the GNSDK Package

GNSDK provides the following modules for application development. For more information about these modules, search this documentation or refer to the table of contents.

GNSDK for Mobile Modules

- DSP
- LOOKUP_LOCALSTREAM
- MANAGER
- MOOD
- MUSICID
- MUSICID_FILE
- MUSICID_STREAM
- PLAYLIST
- STORAGE_SQLITE

Memory Usage

The memory usage of GNSDK for Mobile depends on a number of different factors, including the use of multiple threads, type of recognition, size of a user's collection, number of simultaneous devices connected, metadata requested, and so on. Typical integrations can use anywhere from 5 MB to 30 MB depending on what use cases the application is addressing. There is no "magic number" to describe the memory usage requirements of GNSDK for Mobile.

Memory Usage Guidelines

The following guidelines for RAM are based on measurements of both local and online lookups in a single-threaded environment on the BeagleBoard xM embedded platform. Use of multiple threads, such as with multiple online lookups, can theoretically result in increased maximum memory usage. However, this would require highly unlikely alignment of threads.

Beagle Board xM	
CPU Type	ARM Cortex A8
CPU Speed	1 GHz
CPU Endianness	Little
Storage Size	512 MB
Storage Type	NAND Flash

Beagle Board xM	
RAM size	512 MB
RAM type	LPDDR
OS	Angstrom Linux

RAM Requirements



GNSDK for Mobile may utilize more memory than indicated below. Gracenote always recommends that customers measure heap usage on their device running their application code and utilizing Gracenote's memory usage callback functionality.

Measured RAM Requirements describe the memory usage using the hardware, software and test sets outlined. The measured RAM requirements are for peak usages and will be smaller the majority of time. Recommended RAM Requirements account for future Gracenote feature enhancements.

Feature Set	Measured RAM Requirements*	Recommended RAM Requirements*
MusicID (CD, Text) Music Enrichment (Cover Art)	3 MB	8 MB
MusicID (CD, Text) Music Enrichment (Cover Art) MusicID (File) Playlist, Mood (10K Tracks)	6 MB	14 MB
MusicID (CD, Text) Music Enrichment (Cover Art) MusicID (File) Playlist, Mood (20K Tracks)	9 MB	20 MB
MusicID (CD, Text) Music Enrichment (Cover Art) MusicID (File) Playlist, Mood (40K Tracks)	12 MB	32 MB

*MusicID (Text) lookups were done with a 5K track sample set in the feature set without Playlist and with 10K, 20K and 40K tracks for the corresponding Playlist size. MusicID (CD) lookup measurements were done using a 25K sample set of CD TOCs. MusicID (File) fingerprint lookups were done using 167 tracks from 11 albums.

Playlist Memory Usage

Memory usage for Playlist is directly correlated to the number of tracks available for playlisting. For smaller libraries, RAM requirements may be lower. For libraries with a larger number of songs, RAM requirements may increase. Memory usage increases at a rate of ~0.3 MB per thousand tracks.

Playlist relies on a Collection Summary of stored attribute data of a user's music collection. These collection summaries are designed to take up a minimal amount of disk space and are directly proportional to the user's collection size. Developers should take into account disk storage to persist these collection summaries which grow at the rate of ~110 KB per thousand tracks.

Android

Setting Up the Development Environment

The Mobile sample application requires an Android development environment. To run the Mobile sample applications, you will need to install Google APIs Level 10 or later for Android. These APIs are only used for the samples, and are not necessary for general development. See

<http://developer.android.com/sdk/adding-components.html> for more information. For a description of the Android Developers Tools see: <http://developer.android.com/tools/index.html>

To install the Android SDK, follow the instructions at <http://developer.android.com/about/start.html>

When setting up the environment is complete, the following components, tools, variables, and plug-ins should be installed on the build system.

- Android ARMv6 or ARMv7 based phone
- Android PPlatform Version 2.3.3 (API Level 10) or higher
- Google APIs by Google Inc. (Android API Level 10). Required for the sample application only.
- Android Studio
- JDK 5 or newer (installing JRE alone is not sufficient):
<http://www.oracle.com/technetwork/java/javase/downloads/index.html>
- Android SDK tools directory, and the platform-tools directory, in your PATH environment variable



Gracenote uses GCC 4.6 from NDK 8e. This produces native libraries that are incompatible with other native libraries built with GCC 4.4.3 (the oldest Android compiler). If you incorporate your own native libraries into your applications, you must use GCC 4.6 or higher to build them.



If you are using C++ or Java in your Android development, you must link the libgabi++_shared.so library into your application. This library is available in the Android NDK beginning with version 8d.



When developing on a Windows platform, be sure to download the USB drivers package. These are required for the development environment to communicate with a physical Android device.

Mobile Deployment

GNSDK is delivered as an Android Studio project which can be directly imported into your IDE.



Before continuing, you should already have run the Sample Application as detailed in the *Getting Started with the Android Sample Application*. This document contains details about setting up your Android development environment.

Integrating into an Existing Android Project

The GNSDK can be integrated into an existing Android project. It is distributed as the following libraries:

- gnsdk.jar (Java library)
- gnsdk_helpers.jar (Java library)
- libgnm_decoder.<version number>.so (native library)
- libgnsdk_dsp.<version number>.so (native library)
- libgnsdk_java_marshall.so (native library)
- libgnsdk_lookup_local.<version number>.so (native library)
- libgnsdk_lookup_localstream.<version number>.so (native library)
- libgnsdk_manager.<version number>.so (native library)
- libgnsdk_moodgrid.<version number>.so (native library)
- libgnsdk_musid.<version number>.so (native library)
- libgnsdk_musid_file.<version number>.so (native library)
- libgnsdk_musid_stream.<version number>.so (native library)
- libgnsdk_playlist.<version number>.so (native library)
- libgnsdk_storage_sqlite.<version number>.so (native library)
- libgabi++_shared.so (native Android library)

Ensure that these files can be accessed by your existing Android project and are located by your application build system. Gracenote libraries are not intended to be copied into the Android system folders (root). All of the Gracenote libraries should be kept with the application code (apk). This is essential for application upgradability and multi-application support.

GNSDK Android Permissions

To use the GNSDK properly in your Android application, it must be configured with specific permissions, including:

- Record audio
- Access fine (GPS) location (used only by the sample, not the SDK)
- Write to external storage
- Access Internet
- Access network state

These permissions must be added to the Android application's AndroidManifest.xml file. For a complete list of required permissions and an example of how they are defined, see the Sample Application's AndroidManifest.xml file.

Advanced Implementations

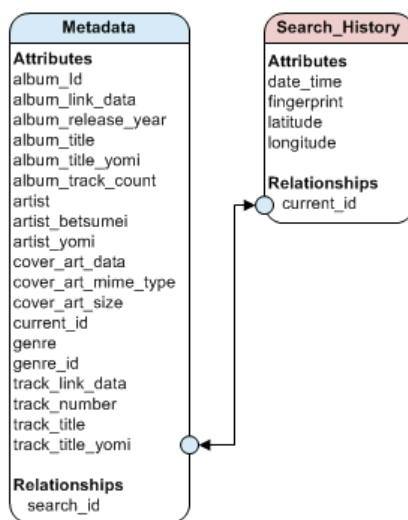
User History

The GNSDK Sample Application includes a user history feature. It stores the results of certain operations and allows the user to view them. When viewed, the history record displays the recognized music, including Cover Art and the location of the user when the operation occurred.

The feature uses a local SQLite database to store the user history information.

Data Model

A simple data model is used to store the data as shown below.



Sample Application User History Database Data Model

The Sample Application includes a history database, and uses a helper class, called `DataBaseAdapter`, to handle it. It creates a simplified interface for managing the database and adding, deleting, and retrieving data from the database.

Internally, `DataBaseAdapter` uses Android classes that simplify interacting with a SQLite database. The `SQLiteDatabase` class provides methods for creating a database, deleting a database, and issuing SQL queries. `SQLiteDatabase` can be used with another helper class, `SQLiteOpenHelper`, which uses a transaction-based method to interact with the underlying SQL database to ensure it is always in a usable state.

Internally, `DataBaseAdapter` uses `DatabaseHelper`, which extends `SQLiteOpenHelper` and implements the `onCreate`, `onOpen`, and `onUpgrade` methods. When a database is created, `onCreate` and `DatabaseHelper` use SQL queries to create the database tables. The SQL queries are used to implement the user history data model.

Adding Entries

When results for an appropriate operation are received, an entry is added to the database. Only results from the following operations are stored:

- Recognizing music from the microphone (IDNow)
- Recognizing music from a PCM sample (AlbumID)

The process of adding an entry to the database is initiated during the callback process. The process retrieves the current GPS location information from the device and submits the results to the database using the `DataBaseAdapter.insertChanges` method.

Limiting Database Size

The size of the database cannot be allowed to grow indefinitely. After adding an entry to the database, its size is checked to determine if it exceeds a predefined limit of 1000 entries. If there are more than 1000 entries, the oldest entries in the database are deleted, bringing the size back to the predetermined limit.

Recalling Entries

The Sample Application uses an SQL query to retrieve all the rows in the database. These are returned as a Cursor object that allows each value in each row returned to be extracted using getter methods. The sample application extracts the rows into Locations objects that are used to populate a UI that allows the user to navigate the entries. For large databases, a paging mechanism can be used to reduce the number of rows exported into memory at any one time.

Mobile Getting Started

Getting Started with Mobile Android

Introduction

GNSDK for Mobile (Android) provides a Sample Application that demonstrates basic functionality. The SDK also provides a development project that is an example of how to incorporate GNSDK for Mobile into your Android application.

This document describes how to integrate the Sample Application project into your development environment.

For additional samples, contact your Gracenote representative.

Fingerprint and Metadata Samples

Gracenote provides a local cache of fingerprints and metadata of some sample songs. The SDK uses this data to attempt a local ID prior to attempting an online lookup. The SDK provides song samples in the sample_music folder that you can use to test local lookup identification.

For additional samples, contact your Gracenote representative.

Artist	Track	Album
Winterwood	Saturday	Love In The Heart
Winterwood	We Never Take The Time	Homeward Tonight

Setting Up the Development Environment

The Mobile sample application requires an Android development environment. To run the Mobile sample applications, you will need to install Google APIs Level 10 or later for Android. These APIs are only used for the samples, and are not necessary for general development. See <http://developer.android.com/sdk/adding-components.html> for more information. For a description of the Android Developers Tools see: <http://developer.android.com/tools/index.html>

To install the Android SDK, follow the instructions at <http://developer.android.com/about/start.html>

When setting up the environment is complete, the following components, tools, variables, and plug-ins should be installed on the build system.

- Android ARMv6 or ARMv7 based phone
- Android PPlatform Version 2.3.3 (API Level 10) or higher
- Google APIs by Google Inc. (Android API Level 10). Required for the sample application only.

- Android Studio
- JDK 5 or newer (installing JRE alone is not sufficient):
<http://www.oracle.com/technetwork/java/javase/downloads/index.html>
- Android SDK tools directory, and the platform-tools directory, in your PATH environment variable



Gracernote uses GCC 4.6 from NDK 8e. This produces native libraries that are incompatible with other native libraries built with GCC 4.4.3 (the oldest Android compiler). If you incorporate your own native libraries into your applications, you must use GCC 4.6 or higher to build them.



If you are using C++ or Java in your Android development, you must link the libgabi++_shared.so library into your application. This library is available in the Android NDK beginning with version 8d.



When developing on a Windows platform, be sure to download the USB drivers package. These are required for the development environment to communicate with a physical Android device.

Setting Up a Device

After setting up your Android development environment, you will need to set up one or more Android devices. For complete instructions, see the Android Developers Guide:

<http://developer.android.com/guide/developing/device.html>

When using a physical device, ensure that it has the following options checked:

- Unknown Sources on the Applications menu
- USB Debugging on the Development menu

Creating the Sample Application

Gracernote provides an Android Studio project for the Sample Application that you can import into your Android development environment. The project incorporates the Gracernote libraries and Sample Application source code.

1. Extract the files from the GNSDK for Mobile package to a location on your development machine. For example, extract the files to `C:\GN_Music_SDK_Android_<version>`, where `<version>` is the current release number.
2. Write-enable the extracted folders and files so they can be updated by the build process.
3. Launch Android Studio and select **Open an existing Android Studio Project**
4. Browse to the location where you extracted the GNSDK for Mobile package (step 1)
The project should now be created in your development environment.

Music Recognition Widget

The SDK also comes with a Widget that can be used for music recognition. You can install this on your Android device's home page through the normal mechanism specific to your device.

Adding Client ID and License Information to the Sample Application

Before you can build and run the Sample Application, you must update the application to use your Gracenote Client ID and license information.

1. In Android Studio, open the Sample Application source file:
`src/com/customer/example/GracenoteMusicID.java`.
2. Add Client ID information:
 - a. Locate the `gnsdkClientID` declaration in the `GracenoteMusicID` class.
 - b. Set the value of `gnsdkClientID` to the six digit Client ID, for example: 123456.
 - c. Locate the `gnsdkClientTag` declaration on the following line of code.
 - d. Set the value of `gnsdkClientTag` to the Client ID Tag, for example: 789123456789012312.
3. Add license information:
 - a. Locate the `gnsdkLicenseFilename` declaration on the line of code following the `gnsdkClientTag` declaration.
 - b. Place the license text file in the `assets` folder of your project directory, and set the value of `gnsdkLicenseFilename` to the name of that file. For example, `mylicensefile.txt`.

Building and Running the Sample Application

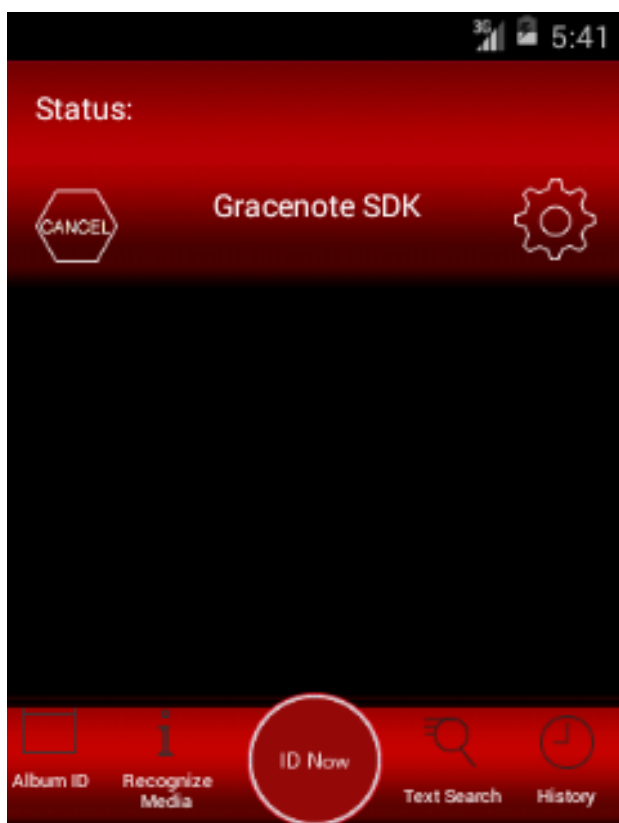
To build and run the Sample Application:

1. Make sure you have a connected device (hardware) to your development environment. If no devices are connected, Android Studio launches the first available virtual device.

For hardware devices:

- a. Connect the device to a USB port.
 - b. Ensure that the device configuration enables the Enable USB Debugging and Unknown Sources options.
2. In Android Studio, choose **Run>Run**.
 3. In the **Run As** dialog box, choose **Android Application (app)**. This builds the application and deploys it to your connected device.

4. The application will start running. It will appear similar to the image below:



5. Confirm the application works: place your device near an audio music source and click the id Now button. If no results are returned, then the application could not identify the audio files. If this is the case, verify the audio is sufficiently loud and repeat the test.

Implementing Applications (All Platforms)

About the Implementing Applications Documentation

This section is intended as a general guide for applications using the object-oriented GNSDK, which supports development in a number of object-oriented languages such as C++, C#, Java, Android Java and Objective-C. This document discusses classes and methods in a generic sort of way while showing brief inline code snippets in different languages. For your particular language, some naming could be slightly different. Java, for example, uses camel casing (first letter is lower-case). This is a small difference, but for other languages, like Objective-C, the differences could be more involved. To help with this, dropdown code samples in specific programming languages are provided.

For specific programming language implementation details, see the language's respective API reference documentation.

Basic Application Steps

To get started with GNSDK development, Gracenote recommends you follow these general steps, some of which are required and some of which are optional but recommended:

1. Required initialization—see *"Setup and Initialization"* :
 - **Get authentication**—Get a Client ID/Tag and license file from Gracenote (GSS). These are used for initial authorization and in every query.
 - **Include necessary header files**—Include the header files and classes for your platform that your application requires.
 - **Initialize SDK** —Instantiate and initialize a GNSDK Manager object (`GnManager` class).
 - **Get a User Object** —Instantiate and initialize a User object (`GnUser`). All queries require a User object with correct Client ID information. You can create a new User, or deserialize an existing User, to get a User object.
2. Not required but suggested initialization:
 - **Enable logging**—Gracenote recommends that GNSDK logging be enabled to some extent. This aids in debugging applications and reporting issues to Gracenote. See *"Using Gracenote SDK Logging"* for more information.
 - **Load a locale(s)**—Gracenote recommends using locales as a convenient way to group locale-dependent metadata specific to a region (such as Europe) and language that should be returned from the Gracenote service. In addition, some metadata requires having a locale loaded. See *"Loading a Locale"* for more information.
 - **Enable GNSDK storage and caching**—See *"Enabling and Using GNSDK Storage"* for

more information. This is not required (unless you are doing local lookups), but highly recommended.

3. If you are using a local database of fingerprints and metadata for identification, then download it and save it to local storage. Set the appropriate mode for online and local lookups. See "Using a Local Fingerprint Database" and "Setting Local and Online Lookup Modes" for more information.
4. Perform queries to identify music. See "Identifying Music"
5. Parse query result objects for desired metadata. See "Processing Returned Metadata Results" for more information.
6. Exit the application when done, no cleanup necessary.

Setup and Initialization

To get started with GNSDK development:

1. Get a Client ID/Tag and License file for application authentication from Gracenote. See [Authorizing a GNSDK Application](#).
2. Include the GNSDK header for your platform to include all necessary libraries and headers. See [Including Header Files](#).
3. Instantiate a `GnManager` object. See [Instantiating a GNSDK Manager Object](#).
4. Instantiate a `GnUser` user object. See [Instantiating a User Object](#).

Authorizing a GNSDK Application

Gracenote uses product licensing and server-side entitlements to manage your application's access to metadata.

As a Gracenote customer, Gracenote works with you to determine the kind of products you need (such as MusicID, Playlist, and so on). Gracenote also determines the metadata your application requires for the products you license.

Gracenote uses this information to create a unique customer ID (called a Client ID/Tag), a license file, and server-side metadata entitlements specifically tailored to your application.

When developing a GNSDK application, you must include a Client ID and license file to authorize your application with Gracenote. In general, the License file enables your application to use the Gracenote products (and their corresponding GNSDK modules) that you purchased. Gracenote Media Services uses the Client ID to enable access to the metadata your application is entitled to use.

All applications are entitled to a set of core metadata based on the products that are licensed. Your application can access enriched metadata through server-side metadata entitlements. Contact your Gracenote representative for more information.



Some applications require a local (embedded) database for metadata. These systems do not access Gracenote Media Services to validate metadata entitlements and access metadata. Instead, metadata entitlements are pre-applied to the local database.

Client ID/Tag

Each GNSDK customer receives a unique client ID/Tag string from Gracenote. This string uniquely identifies each application to Gracenote Media Services and lets Gracenote deliver the specific metadata the application requires.

A Client ID/Tag string consists of two sets of numbers separated with a hyphen. The number before the hyphen is considered the 'ID' and the number after, the 'Tag'. A client ID/Tag string has the following format:

```
<10 character client ID>-<17 character client ID tag>
```

License File

Gracenote provides a license file along with your Client ID. The license file notifies Gracenote to enable the GNSDK products you purchased for your application.



You should secure your Gracenote client id, tag and license information. Something similar to the way Android recommends protecting a Google Play public key from malicious hackers: http://developer.android.com/google/play/billing/billing_best_practices.html

Including Header Files

GNSDK consists of a set of shared modules. The GNSDK Manager module is required for all applications. All other modules are optional. Your application's feature requirements determine which additional modules should be used.

For convenience, all your application has to do is include a single GNSDK header file and all necessary header files and libraries will be automatically included.

Java

```
import com.gracenote.gnsdk.*;
```

Objective-C

```
#import <GnSDKObjC/Gn.h>
```

Windows Phone C#

```
using Gracenote;
```

Instantiating a GNSDK Manager Object

The first thing your application needs to do is initialize an SDK Manager object (`GnManager`) using the GNSDK library path and the contents of the license file you obtained from GSS. The SDK Manager object is used to monitor an application's interaction with Gracenote.

Specifying the License File

Your application must provide the license file when you allocate a `GnManager` object. This class' constructor gives you the following options for submitting the license file:

- **Null-terminated string**—Set the input mode parameter to `GnLicenseInputMode.kLicenseInputModeString` and pass the license file as a null-terminated string (see examples below).
- **Filename**—Set the input mode parameter to `GnLicenseInputMode.kLicenseInputModeFilename` and pass the relative filename in the string parameter.

Code samples

Android Java

```
private String gnsdkLicense = <get license as string from asset>;
Context context = this.getApplicationContext();

// Initialize GNSDK
GnManager gnsdk = new GnManager(context, // Android Context
                                gnsdkLicense, // License as a string
                                GnLicenseInputMode.kLicenseInputModeString); // Input License as a string
```

Objective-C

```
@property (strong) GnManager *gnManager;
NSError* error = nil;
NSString* resourcePath = [[NSBundle mainBundle] pathForResource:
                          gnsdkLicenseFilename ofType: nil];
NSString* licenseString = [NSString stringWithContentsOfFile: resourcePath
                      encoding: NSUTF8StringEncoding
                      error: &error];
self.gnManager = [[GnManager alloc] initWithLicense: licenseString licenseInputMode:
kLicenseInputModeString];
```

Windows Phone C#

```
/*
 * Initialize GNSDK
 */
string licenseString = ReadFile(App.gnLicenseFileName_);
App.gnManager_ = new GnManager(licenseString, GnLicenseInputMode.kLicenseInputModeString);
```

Instantiating a User Object

To make queries, every application is required to instantiate a User object (`GnUser`). Most devices will only have one user; however, on a server, for example, there could be a number of users running your application. Gracenote uses the Client ID and Client Tag to verify that the licensed and allowable users quota has not been exceeded.

If an application cannot go online at user-registration, but has a local fingerprint database it can use for identification, it has the option to create and use a 'local only' User object. A User object created with online connectivity can do both local and online queries. To determine if the User object can only do local queries, your application can call the `GnUser`'s `IsLocalOnly` method.

`GnUser` objects can be created 'online,' which means the Gracenote back-end creates and verifies them. Alternatively, they can be created 'local only,' which means the SDK creates and uses them locally.

For example (Java):

```
// Create user for video and music
gnUser = new GnUser(gnUserStore, clientId, clientTag, "1.0");
```

Saving the User Object to Persistent Storage

User objects should be saved to persistent storage. If an app registers a new user on every use instead of retrieving it from storage, then the user quota maintained for the Client ID is quickly exhausted. Once the quota is reached, attempting to create new users will fail. To maintain an accurate usage profile for your application, and to ensure that the services you are entitled to are not being used unintentionally, it is important that your application registers a new user only when needed, and then stores that user for future use.

To save to persistent storage, you have the option to implement the `IGnUserStore` interface which requires you to implement two methods: `LoadSerializedUser` and `StoreSerializedUser`.



On mobile and ACR platforms (Android, iOS, Windows), the SDK provides the `GnUserStore` class, a platform-specific implementation of the `IGnUserStore` interface. Storage on these devices is implemented in platform-specific ways. On Android, for example, the User object is saved to shared preferences.

You can then pass an instance of this class as a parameter in a `GnUser` constructor and the SDK will automatically read the User object from storage and use it to create a new User ID. The SDK may also periodically `SaveSerializedUser` when user data changes.

Allocating a User object code samples:

Android Java

```
// Provide Android Context to GnUserStore
GnUser gnUser = new GnUser( new GnUserStore(context), clientId, clientTag, appString );
```

Objective-C

```
@property (strong) GnUser *gnUser;
self.gnUserStore = [[GnUserStore alloc] init];
self.gnUser = [[GnUser alloc] initWithGnUserStoreDelegate: self.gnUserStore
                      clientId: clientId
                      clientTag: clientIDTag
                      applicationVersion: @"1.0.0.0"];
```

Windows Phone C#

```
// Create ACR User
if (null == App.gnUser_)
{
    App.gnUser_ = new GnUser(new GnUserStore(App.gnClientId_),
                            App.gnClientId_,
                            App.gnClientIdTag_,
                            App.applicationVersion_);
    App.gnUser_.Options.LookupMode(GnLookupMode.kLookupModeOnline);
}
```

Loading a Locale

GNSDK provides *locales* as a convenient way to group locale-dependent metadata specific to a region (such as Europe) and language that should be returned from the Gracenote Service. A locale is defined by a group (such as Music), a language, a region and a descriptor (indicating level of metadata detail), which are identifiers to a specific set of lists in the Gracenote Service.

Using locales is relatively straightforward for most applications to implement. However, it is not as flexible or complicated as accessing lists directly - most locale processing is handled in the background and is not configurable. For most applications though, using locales is more than sufficient. Your application should only access lists directly if it has a specific reason or use case for doing so.

To load a locale, allocate a `GnLocale` object with one of the class constructors. The `GnLocale` constructors take parameters indicating the following:

- **Group** - Locale group type such as Music, Playlist, EPG or Video that can be easily tied to the application's use case
- **Region** - Region the application is operating in, such as US, China, Japan, Europe, and so on, possibly specified by the user configuration
- **Language** - Language the application uses, possibly specified by the user configuration
- **Descriptor** - Additional description of the locale, such as Simplified or Detailed for the list hierarchy group to use, usually determined by the application's use case
- **Status callback (optional)** - One of the constructors takes a `GnStatusEventsListener` callback object



Locales have the following space requirements: 2MB for a music only locale, 6MB for a music and playlist locale.

For example:

- A locale defined for the USA of English/ US/Detailed returns detailed content from a list written in English for a North American audience.

- A locale defined for Spain of Spanish/Global/Simplified returns list metadata of a less-detailed nature, written in Spanish for a global Spanish-speaking audience (European, Central American, and South American).

Java/Android Java

```
GnLocale locale =
    new GnLocale(GnLocaleGroup.kLocaleGroupMusic,
        GnLanguage.kLanguageEnglish,
        GnRegion.kRegionGlobal,
        GnDescriptor.kDescriptorDefault,
        gnUser);

locale.setGroupDefault();
```

Objective-C

```
GnLocale *locale =
    [[GnLocale alloc] initWithGnLocaleGroup: kLocaleGroupMusic
        language: kLanguageEnglish
        region: kRegionGlobal
        descriptor: kDescriptorSimplified
        user: self.gnUser
        statusEventsDelegate: nil];

[locale setGroupDefault:&localeError];
```

Windows Phone C#

```
GnLocale locale =
    new GnLocale(GnLocaleGroup.kLocaleGroupMusic,
        GnLanguage.kLanguageEnglish,
        GnRegion.kRegionGlobal,
        GnDescriptor.kDescriptorDefault,
        App.mGnUser,
        null
    );

locale.SetGroupDefault();
```

Locale-Dependent Data

The following metadata fields require having a locale loaded:

- artist type - levels 1-2
- audience
- era - levels 1-3
- genre - levels 1-3
- mood - levels 1-2
- origin - levels 1-4
- tempo - levels 1-3
- rating
- rating description
- rating type
- rating type ID
- reputation

- scenario
- role
- role category
- serial type
- setting environment
- setting time period
- story type
- entity type
- composition form
- instrumentation
- package display language
- EPG level categories
- video feature type
- video production type
- video media type
- video region
- video region description
- video topic
- video work type

Default Regions and Descriptors

When loading a locale, your application provides inputs specifying group, language, region, and descriptor. Region and descriptor can be set to “default.”

When no locales are present in the local database, or no local database is enabled, and the application is configured for online access, GNSDK uses the Global region when the default region is specified, and the Detailed descriptor when the default descriptor is specified.

Otherwise, when “default” is specified, GNSDK filters the local database and loads a locale matching the group and language (and the region and descriptor, if they are not specified as default). Complete locales (those with all sub-components present) are preferred over incomplete locales. If, after filtering, the local database contains multiple equally complete locales, a default locale is chosen using the defaults shown in the table below:

Regional GDB	Available Locales	Default Locale
North America (NA)	US and Latin America	US
Latin America (LA)	Latin America	Latin America
Korea (KR)	Korea	Korea
Japan (JP)	Japan	Japan
Global (GL)	Global, Japan, US, China, Taiwan, Korea, Europe, and Latin America	Global
Europe (EU)	Europe	Europe
China (CN)	China and Taiwan	China

If no locales are present after filtering the local database, an error is returned.

Default regions and descriptors can be used to write generic code for loading a locale. For example, consider an application targeted for multiple devices: one with a small screen, where the Simplified locales are desired; and one with a large screen, where more detail can be displayed to the user, and the Detailed locales are desired. The application code can be written to generically load locales with the “default” descriptor, and each application can be deployed with a local database containing simplified locales (small-screen version), or detailed locales (large-screen version). GNSDK loads the appropriate locales based on the contents of the local database.

Locale Groups

Setting the locale for a group causes the given locale to apply to a particular media group - Music, Playlist, Video or EPG. For example, setting a locale for the Music group applies the locale to all music-related objects. When a locale is loaded, all lists necessary for the locale group are loaded into memory. For example, setting the locale for the Playlist group causes all lists needed to generate playlists to be loaded.

Once a locale has been loaded, you must call the `GnLocale's SetGroupDefault` method before retrieving locale-dependent values.

Multi-Threaded Access

Since locales and lists can be accessed concurrently, your application has the option to perform such actions as generating a Playlist or obtaining result display strings using multiple threads.

Typically, an application loads all required locales at start up, or when the user changes preferred region or language. To speed up loading multiple locales, your application can load each locale in its own thread.

Updating Locales and Lists

GNSDK supports storing locales and their associated lists locally, which improves access times and performance. Your application must enable a database module (such as SQLite) to implement local storage. For more information, See "Enabling and Using GNSDK Storage".

Update Notification

Periodically, your application may need to update any locale lists that are stored locally. As a best practice, Gracenote recommends registering a locale update notification callback or, if you are using lists directly, a lists update notification callback. To do this, you need to code an `IGnSystemEvents` delegate that implements the locale or list update methods—`LocaleUpdateNeeded` or `ListUpdateNeeded`—and provide that delegate as a parameter to the `GnManager's EventHandler` method. When GNSDK internally detects that a locale or list is out of date, it will call the appropriate callback. Detection occurs when a requested list value is not found. This is done automatically without the need for user application input. Note, however, that if your application does not request locale-dependent metadata or missing locale-dependent data, no detection will occur.



Updates require the user lookup mode option to be set to online lookup - `kLookupModeOnline`(default) or online lookup only—`kLookupModeOnlineCacheOnly`. This allows the SDK to retrieve locales from the Gracenote Service. You may need to toggle

this option value for the update process. For more information about setting the user option, "Setting Local and Online Lookup Modes".

Once your app receives a notification, it can choose to immediately do the update or do it later. Gracernote recommends doing it immediately as this allows the current locale/list value request to be fulfilled, though there is a delay for the length of time it takes to complete the update process.



The GNSDK does not set an internal state or persistent flag indicating an update is required; your application is responsible for managing the deferring of updates beyond the notification callback.

Locale Behavior


How locales are stored, accessed and updated depends on how you have configured your storage and lookup options as shown in the following table. For information on configuring lookup modes see "Setting Local and Online Lookup Modes".



Storage Provider Initialized	GnLookupMode Enum	Behavior
Either	<code>kLookupModeOnlineNoCache</code>	Locales are always downloaded and stored in RAM, not local storage.
Not initialized	<code>kLookupModeOnline</code>	Locales are always downloaded and stored in RAM, not local storage.
Initialized	<code>kLookupModeOnline</code>	If downloaded, locales are read from local storage. Downloaded locales are written immediately to local storage.
Not Initialized	<code>kLookupModeOnlineNoCacheRead</code>	Locales are always downloaded and stored in RAM.
Initialized	<code>kLookupModeOnlineNoCacheRead</code>	Locales are always downloaded and stored in RAM and local storage. Locale data is always read from RAM, not local storage.
Initialized	<code>kLookupModeOnlineCacheOnly</code> <code>kLookupModeLocal</code>	Locale data is read from local storage. If requested data is not in locale storage the load attempt fails. Local storage is updated when new versions become available. The application developer is responsible for providing that mechanism.



Locale behavior may not change if the lookup mode is changed after the locale is loaded. For example, if a locale is loaded when the lookup mode is `kLookupModeOnline`, locale data will be read from local storage even if the lookup mode is changed.

Best Practices

Practice	Description
Applications should use locales.	Locales are simpler and more convenient than accessing lists directly. An application should only use lists if there are specific circumstances or use cases that require it.
Apps should register a locale/list update notification callback and, when invoked, immediately update locales/lists.	See the Update Notification section above.
Applications can deploy with pre-populated list stores and reduce startup time.	<p>On startup, a typical application loads locale(s). If the requested locale is not cached, the required lists are downloaded from the Gracenote Service and written to local storage. This procedure can take time.</p> <p>Customers should consider creating their own list stores that are deployed with the application to decrease the initial startup time and perform a locale update in a background thread once the application is up and running.</p>
Use multiple threads when loading or updating multiple locales.	Loading locales in multiple threads allows lists to be fetched concurrently, reducing overall load time.
Update locales in a background thread.	<p>Locales can be updated while the application performs normal processing. The SDK automatically switches to using new lists as they are updated.</p> <div>  <p>If the application is using the GNSDK Manager Lists interface directly and the application holds a list handle, that list is not released from memory and the SDK will continue to use it.</p> </div>
Set a persistence flag when updating. If interrupted, repeat update.	<p>If the online update procedure is interrupted (such as network connection/power loss) then it must be repeated to prevent mismatches between locale required lists.</p> <p>Your application should set a <i>persistence</i> flag before starting an update procedure. If the flag is still set upon startup, the application should initiate an update. You should clear the flag after the update has completed.</p>

Practice	Description
Call the <code>GnStoreOps'</code> <code>Compact</code> method after updating lists or locales.	<p>As records are added and deleted from locale storage, some storage solutions, such as SQLite, can leave empty space in the storage files, artificially bloating them. You can call the <code>GnStoreOps'</code> <code>Compact</code> method to remove these.</p> <div>  <p>The update procedure is not guaranteed to remove an old version of a list from storage immediately because there could still be list element references which must be honored until they are released. Therefore, your application should call the <code>GnStoreOps'</code> <code>Compact</code> method during startup or shutdown after an update has finished.</p> </div>
Local only applications should set the user handle option for lookup mode to local only.	<p>If your application wishes to only use the Locales in pre-populated Locales storage, then it must set the user handle lookup mode to local.</p> <p>For example (C++)</p> <pre>/* Set lookup mode (all queries done with this user will inherit the lookup mode) */ user.Options().LookupMode(kLookupModeLocal);</pre>
To simplify the implementation of multi-region applications, use the default region and descriptor.	<p>The Locale subsystem can infer a region and descriptor from the Locale store that can be used in place of the region and descriptor defaults when loading a locale. This can simplify implementing an application intended to be deployed in different regions with its own region specific pre-populated Locale store.</p> <div>  <p>If you are deploying your app to multiple regions with a pre-populated Locale store containing locales for all target regions then you should use <code>kRegionDefault</code> and <code>kDescriptorDefault</code> when loading a locale. In this case, the same region and descriptor are used based on defaults hardcoded into the SDK.</p> </div>

Using Gracernote SDK Logging

The `GnLog` class has methods to enable Gracernote SDK logging, set options, write to the SDK log, and disable SDK logging.

There are 3 approaches you can take to implementing logging using the GNSDK:

1. **Enable GNSDK logging**—This creates log file(s) that you and Gracernote can use to evaluate and debug any problems your application might encounter when using the SDK
2. **Enable GNSDK logging and add to it**—Use the `GnLog Write` method to add your application's log entries to the GNSDK logs.
3. **Implement your own logging mechanism (via the logging callback)**— Your logging callback, for example, could write to the console, Unix Syslog, or the Windows Event Log.



Gracernote recommends you implement callback logging (see [Implementing Callback Logging](#)). On some platforms, for example, Android, GNSDK logging can cause problematic system delays. Talk to your Gracernote representative for more information.

Enabling GNSDK Logging

To use Gracernote SDK logging:

1. Instantiate a `GnLog` object.

This class has two constructors: both require you to set a log file name and path, and a `IGnLogEvents` logging callback delegate (`GnLogEventsDelegate` in Objective-C). One of the constructors also allows you to set logging options, which you can also set via class methods. These include ones for:

- What type of messages to include: error, warning, information or debug (`GnLogFilters`)
 - What fields to log: timestamps, thread IDs, packages, and so on. (`GnLogColumns`)
 - Maximum size of the log file in bytes, synchronous or asynchronous logging, and archive options (`GnLogOptions`)
2. Call the `GnLog Enable (PackageID)` method to enable logging for specific packages or all packages.



A *package* is a GNSDK Library as opposed to a module, which is a block of functionality within a package. See the `GnLogPackageType` enums for more information on GNSDK packages.



Note that `Enable` returns its own `GnLog` object to allow method chaining.

3. Call the `GnLog Write` method to write to the GNSDK log

4. Call the `GnLog Disable (PackageID)` method to disable logging for a specific package or all packages.

Logging code samples

Java

```
// Enable GNSDK logging
String gracenoteLogFilename = Environment.getExternalStorageDirectory().getAbsolutePath() +
File.separator + gnsdkLogFilename;
gnLog = new GnLog(gracenoteLogFilename, null);
gnLog.columns(new GnLogColumns().all());
gnLog.filters(new GnLogFilters().all());
gnLog.enable(GnLogPackageType.kLogPackageAll);
```

Objective-C

```
NSString *docsDir = [GnAppDelegate applicationDocumentsDirectory];
docsDir = [docsDir stringByAppendingPathComponent:@"log.txt"];

self.gnLog = [[GnLog alloc] initWithLogFilePath:docsDir
filters:[[[GnLogFilters alloc] init]all]
columns:[[[GnLogColumns alloc] init]all]
options:[[[GnLogOptions alloc] init]maxSize:0]
logEventsDelegate:self];

// Max size of log: 0 means a new log file will be created each run
[self.gnLog options: [[GnLogOptions alloc] init]maxSize:0]];
[self.gnLog enableWithPackage:kLogPackageAllGNSDK error:nil];
```

C#

```
/* Enable GNSDK logging */
App.gnLog_ = new GnLog(Path.Combine(Windows.Storage.ApplicationData.Current.LocalFolder.Path,
"sample.log"), (IGnLogEvents)null);

App.gnLog_.Columns(new GnLogColumns().All);
App.gnLog_.Filters(new GnLogFilters().All);

GnLogOptions options = new GnLogOptions();
options = options.MaxSize(0);
options = options.Archive(false);
App.gnLog_.Options(options);

App.gnLog_.Enable(GnLogPackageType.kLogPackageAll);
```

The GNSDK logging system can manage multiple logs simultaneously. Each call to the enable API can enable a new log, if the provided log file name is unique. Additionally, each log can have its own filters and options.

Implementing Callback Logging

You also have the option to direct GNSDK to allow a logging callback, where you can determine how best to capture and disseminate specific logged messages. For example, your callback function could write to its own log files or pass the messages to an external logging framework, such as the console, Unix Syslog, or the Windows Event Log.

Enabling callback is done with the `GnLog` constructor where you have the option to pass it a `IGnLogEvents` (`GnLogEventsDelegate` in Objective-C) callback, which takes callback data, a package ID, a filter mask, an error code, and a message field.

Enabling and Using GNSDK Storage

To improve performance, your application can enable internal GNSDK storage and caching. The GNSDK has two kinds of storage, each managed through a different class:

1. **Online stores for lookups**—The GNSDK generates these as lookups take place. Use `GnStoreOps` methods to manage these.
2. **Local lookup databases**—Gracenote Global Support & Services (GSS) generates these databases, which differ based on region, configuration, and other factors, and ships them to customers as read-only files. These support TUI, TOC and text lookup for music searches. The `GnLookupLocal` class can be used to manage these databases.

To enable and manage GNSDK storage:

1. Enable a *storage provider* (SQLite) for GNSDK storage
2. Allocate a `GnManager` object for online stores (optional)
3. Allocate a `GnLookupLocal` object for local lookup databases (optional)
4. Set a folder location(s) for GNSDK storage (required)
5. Manage storage through `GnManager` and `GnLookupLocal` methods

Enabling a Provider for GNSDK Storage

Before GNSDK storage can take place, you need to enable a storage provider. Right now, that means using the GNSDK SQLite module. Note that this is for GNSDK use only—your application cannot use this database for its own storage.



* For information on using SQLite, see <http://sqlite.org>.

* Note that enabling SQLite prevents linking to an external SQLite library for your own use.

In the future, other database modules will be made available, but currently, the only option is SQLite.

To enable local storage, you need to call the `GnStorageSqlite`'s `Enable` method which returns a `GnStorageSqlite` object.

C++

```
/* Enable StorageSQLite module to use as our database engine */  
GnStorageSqlite& storageSqlite = GnStorageSqlite::Enable();
```

Java

```
GnStorageSqlite gnStorageSqlite = GnStorageSqlite.enable();
```

Objective-C

```
self.gnStorageSqlite = [GnStorageSqlite enable: &error];
```

Windows Phone C#

```
App.gnStorageSqlite_ = GnStorageSqlite.Enable;
```

GNSDK Stores

Once enabled, the GNSDK manages these stores:

Stores	Description
Query store	The query store caches media identification requests
Lists store	The list store caches Gracenote display lists
Content store	The content stores caches cover art and related information

You can get an object to manage these stores with the following `GnManager` methods:

- `GnStoreOps& QueryCacheStore`—Get an object for managing the query cache store.
- `GnStoreOps& LocalesStore`—Get an object for managing the locales/lists store.
- `GnStoreOps& ContentStore`—Get an object for managing the content store.

GNSDK Databases

Once enabled, the GNSDK manages these databases as the following `GnLookupLocal::GnLocalStorageName` enums indicate:

Database (GnLookupLocal)	Description
<code>kLocalStorageContent</code>	Used for querying Gracenote content.
<code>kLocalStorageMetadata</code>	Used for querying Gracenote metadata.
<code>kLocalStorageTOCIndex</code>	Used for CD TOC searching.
<code>kLocalStorageTextIndex</code>	Used for text searching.

Setting GNSDK Storage Folder Locations

You have the option to set a folder location for all GNSDK storage or locations for specific stores and databases. You might, for example, want to set your stores to different locations to improve performance and/or tailor your application to specific hardware. For example you might want your locale list store in flash memory and your image store on disk.

If no locations for storage are set, the GNSDK, by default, uses the current directory. To set a location for all GNSDK storage, use the `GnStorageSqlite StorageLocation` method.

Use the `GnStoreOps' Location` method or the `GnLookupLocal's StorageLocation` method to set specific store or database locations. `StorageLocation` takes a database enum and a path location

string. To set a store location, you would need to allocate a `GnStoreOps` object for a specific cache using `GnManager` methods and call its `Location` method.

Examples of setting a location for all stores and databases using SQLite

Java

```
gnStorageSqlite.storageLocation(getExternalFilesDir(null).getAbsolutePath());
```

Objective-C

```
[self.gnStorageSqlite storageLocationWithFolderPath:[GnAppDelegate applicationDocumentsDirectory]  
error: &error];
```

Windows Phone C#

```
App.gnStorageSqlite_.StorageLocation(Windows.Storage.ApplicationData.Current.LocalFolder.Path);
```



For Windows CE, an absolute path must be used in setting storage folder location

Getting Local Lookup Database Information

You can retrieve manifest information about your local databases, including database versions, available image sizes, and available locale configurations. Your application can use this information to request data more efficiently. For example, to avoid making queries for unsupported locales, you can retrieve the valid locale configurations contained in your local lists cache.

Image Information

GNSDK provides album cover art, and artist and genre images in different sizes. You can use the `kImageSize` key with the `GnLookupLocal's StorageInfo` method to retrieve available image sizes. This allows you to request images in available sizes only, rather than spending cycles requesting image sizes that are not available.

Use the `GnLookupLocal's StorageInfoCount` method to provide ordinals to the `StorageInfo` method to get the image sizes.

Database Versions

To retrieve the version number for a local database, use the `kGDBVersion` key with the `StorageInfo` method. Use an ordinal of 1 to get the database version.

C++

```
gnsdk_cstr_t gdb_version = gnLookupLocal.StorageInfo(kMetadata, kGDBVersion, ordinal);
```

Getting Available Locales

Use the `GnLocale's LocalesAvailable` method to get valid locale configurations available in your local lists store. Locale configurations are combinations of values that you can use to set the locale for your application. This method returns values for group, region, language and descriptor. Returns a count of the values available for a particular local database and local storage key.

Setting Online Cache Expiration

You can use the `GnUser's CacheExpiration` method to set the maximum duration for which an item in the GNSDK query cache is valid. The duration is set in seconds and must exceed one day (> 86400). Setting this option to a zero value (0) causes the cache to start deleting records upon cache hit, and not write new or updated records to the cache; in short, the cache effectively flushes itself. The cache will start caching records again once this option is set to a value greater than 0. Setting this option to a value less than 0 (for example: -1) causes the cache to use default expiration values.

Managing Online Cache Size and Memory

You can use the following `GnStorageSqlite` methods to manage online cache size on disk and in memory:

- **MaximumSizeForCacheFileSet**—Sets the maximum size the GNSDK cache can grow to in kilobytes; for example "100" for 100 Kb or "1024" for 1 MB. This limit applies to each cache that is created. If the cache files' current size already exceeds the maximum when this option is set, then the passed maximum is not applied. When the maximum size is reached, new cache entries are not written to the database. Additionally, a maintenance thread is run that attempts to clean up expired records from the database and create space for new records. If this option is not set the cache files default to having no maximum size.

MaximumMemorySizeForCacheSet— Sets the maximum amount of memory SQLite can use to buffer cache data. The value passed for this option is the maximum number of Kilobytes of memory that can be used. For example, "100" sets the maximum to 100 KB, and "1024" sets the maximum to 1 MB.
-

Using a Local Fingerprint Database

Gracenote provides a mechanism for you to use a local database of track fingerprints and metadata for identification. The SDK can use this database to attempt a local ID prior to going online. Doing this provides a significant performance improvement if a local match is found.

To create a local database, you need to download and *ingest* raw fingerprint and metadata from Gracenote, packaged in an entity called a *bundle*. Currently, you have to work with Gracenote Global Support & Services (GSS) to create a bundle. In the future, Gracenote will provide a self-service tool to do this. Having a local database requires that a storage provider, such as SQLite be enabled. You can do this with the `GnStorageSqlite` class.

Your application can ingest multiple bundles. If the same track exists in multiple ingested bundles it is added to the local database only once with the most recent/up-to-date track information. The ingestion process can be lengthy; your application may want to do this on a background thread to avoid stalling the main application thread.

To implement local lookup, use the following `GnLookupLocalStream` methods :

- **Enable** —Enables the `LookupLocalStream` library.
- **StorageLocation** —Sets the location for your MusicID-Stream fingerprint database . Bundle ingestion creates the database at the location specified. If the path does not exist, ingestion will fail.
- **StorageRemove** —Removes an item from the local fingerprint database.
- **StorageClear** —Clears all tracks from the local fingerprint database.

And the following `GnLookupLocalStreamIngest` methods:

- **Write** —Takes byte data and writes specified number of bytes to local storage.
- **Flush** —Flushes the memory cache to the file storage and commits the changes. This method ensures that everything written is committed to the file system. Note that this is an optional call as internally data is flushed when it exceed the cache size and when the object goes out of scope.

Note that the `GnLookupLocalStreamIngest` constructor takes a callback object to handle statuses as the ingest process can take some time. See the API reference documentation for `GnLookupLocalStreamIngest` for more information.

Downloading and Ingesting Bundles

To manage bundles, your application would typically need to do the following:

- Manually retrieve a Gracenote-provided bundle
- Place bundle in an online location that your application can access
- Have your application download and ingest the bundle.
-



Note: Bundles should be retrieved from an online source. Gracenote recommends that when your application is installed or initialized that it download and ingest the latest bundle rather than ship with a bundle as part of the application binaries.

Ingesting bundles code samples:

Android Java

```
// Enable storage provider allowing GNSDK to use its persistent stores
GnStorageSqlite.enable();

// Enable local MusicID-Stream recognition (GNSDK storage provider must be enabled as pre-requisite)
GnLookupLocalStream.enable();

// Ingest MusicID-Stream local bundle, perform in another thread as it can be lengthy
Thread ingestThread = new Thread( new LocalBundleIngestRunnable(context) );
ingestThread.start();

/**
 * Loads a local bundle for MusicID-Stream lookups
 */
class LocalBundleIngestRunnable implements Runnable {
    Context context;

    LocalBundleIngestRunnable(Context context) {
        this.context = context;
    }

    public void run() {
        try {

            // Our bundle is delivered as a package asset
            // to ingest the bundle access it as a stream and write the bytes to
            // the bundle ingester.
            // Bundles should not be delivered with the package as this, rather they
            // should be downloaded from your own online service.

            InputStream bundleInputStream = null;
            int ingestBufferSize = 1024;
            byte[] ingestBuffer = new byte[ingestBufferSize];
            int bytesRead = 0;

            GnLookupLocalStreamIngest ingester = new GnLookupLocalStreamIngest(new BundleIngestEvents
());

            try {

                bundleInputStream = context.getAssets().open("1557.b");

                do {

                    bytesRead = bundleInputStream.read(ingestBuffer, 0, ingestBufferSize);
                    if ( bytesRead == -1 )
                        bytesRead = 0;

                    ingester.write( ingestBuffer, bytesRead );

                } while( bytesRead != 0 );

            }

        }
    }
}
```

```

    } catch (IOException e) {
        e.printStackTrace();
    }

    ingester.flush();

    } catch (GnException e) {
        Log.e( appString, e.errorCode() + ", " + e.errorDescription() + ", " + e.errorModule() );
    }
}
}

```

Objective-C

```

- (NSError *) setupLocalLookup
{
    NSError *    error = nil;

    /*    Initialize the local lookup so we can do local lookup queries.    */
    self.gnLookupLocalStream = [GnLookupLocalStream enable: &error];
    if (! error)
    {
        NSString *    docDir = [GnAppDelegate applicationDocumentsDirectory];
        [self.gnLookupLocalStream storageLocation: docDir
                                error: &error];

        if (! error)
        {
            // Look for the 10,000 track bundle and if not found try the little one.
            NSString*    bundlePath = [[NSBundle mainBundle] pathForResource:@"1557.b" ofType:
nil];

            if (bundlePath)
            {
                [self.gnLookupLocalStream storageClear: &error];

                if (! error)
                {
                    __block GnLookupLocalStreamIngest *lookupLocalStreamIngest =
[[GnLookupLocalStreamIngest alloc] initWithGnLookupLocalStreamIngestEventsDelegate:self];

                    // Load Bundle in a separate thread to keep the UI responsive. This is
required for Large Bundles that can take few minutes to be ingested.

                    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_BACKGROUND,
0), ^{

                        NSError *error = nil;
                        NSInteger bytesRead = 0;
                        double totalBytesRead = 0;
                        uint8_t buffer[1024];
                        NSInputStream *fileInputStream = [NSInputStream
inputStreamWithFileAtPath:bundlePath];

                        [fileInputStream open];

                        do {
                            bytesRead = [fileInputStream read:buffer maxLength:1024];
                            [lookupLocalStreamIngest write:buffer dataSize:sizeof(buffer)
error:&error];

                            if(error)

```

```

        {
            NSLog(@"Error during lookupLocalStreamIngest write: %@", [error
localizedDescription]);
        }

        totalBytesRead+=bytesRead;

        }while (bytesRead>0);

        [lookupLocalStreamIngest flush:&error];
        [fileInputStream close];

    });
}
}
}

return error;
}

```

Windows Phone C#

```

// Ingest local fingerprint bundle
GnLookupLocalStream lookupLocalStream = GnLookupLocalStream.Enable;

lookupLocalStream.StorageLocation(Windows.Storage.ApplicationData.Current.LocalFolder.Path);

GnLookupLocalStreamIngest lookupLocalStreamIngest = new GnLookupLocalStreamIngest(this);

// Bundle asset path including asset name. The code assumes that the bundle
// to be ingested was added to the project as a content file.
string bundlePath = "1557.b";

Stream bundleFileStream = Application.GetResourceStream(new Uri(bundlePath,
UriKind.Relative)).Stream;

byte[] buffer = new byte[1024 + 10];
int count = 1024;
int read = 0;
while (0 != (read = bundleFileStream.Read(buffer, 0, count)))
{
    lookupLocalStreamIngest.Write(buffer, (uint)read);
}

lookupLocalStreamIngest.Flush();

```

Designating a Storage Provider

A *storage provider* module implements GNSDK local storage. There are currently two available GNSDK storage providers: SQLite and QNX's QDB.

By default, no storage provider is enabled. To use SQLite, your application needs to instantiate a `GnStorageSqlite` object and call its `enable` method.

Setting Local and Online Lookup Modes

You can set lookup modes to determine if GNSDK lookups are done locally or online. GNSDK is designed to operate exactly the same way in either case. You can use the `GnUser.LookupMode` method to set this option for the user. You can also set this option for specific queries.

The terms *local* and *online* apply to the following:

1. **Online lookup**—Refers to queries made to the Gracenote service over the Internet.
2. **Online queries stored locally**—The GNSDK generates these as lookups take place. Even though they are stored locally, online stores are considered part of online lookup, not local lookup. The `GnManager` class can be used to manage these stores. Note that this store requires your application to enable GNSDK storage and caching. See *"Enabling and Using GNSDK Storage"* for more information.
3. **Local lookup fingerprint and metadata database**—Gracenote provides a mechanism for using a local database of fingerprints and metadata for identification. The `GnLookupFpLocal` class can be used for this database. See *Using a Local Fingerprint Database* for more information on this topic.

Supported Lookup Modes

GNSDK supports the following lookup mode options as shown with these `GnLookupMode` enums:

- **kLookupModeOnline**—This is the default lookup mode. First, the query checks cache (if it exists) for a match. If no match is found in the cache, then an online query to the Gracenote Service is performed. If a result is found there, it is stored in the local online cache. The query fails if no connection to the Gracenote Service exists. Via the User object, you can set the length of time before cache lookup query expires.
- **kLookupModeLocal**—Forces the lookup to be done against the local database only. Local stores created from (online) query results are not queried in this mode. The query fails if no local database exists.
- **kLookupModeOnlineCacheOnly**—Queries are done against the online cache only and does not perform a network lookup. The query fails if no online provider exists.
- **kLookupModeOnlineNoCache**—Forces the query to be done online only and does not perform a

local cache lookup first. The query fails if no online provider exists. Online queries and lists are not written to local storage, even if a storage provider has been initialized.

- **kLookupModeOnlineNoCacheRead**—Forces the query to be done online only and does not perform a local cache lookup first. The query fails if no online provider exists. Online queries and lists are not written to local storage, even if a storage provider has been initialized.

The local and online modes are the standard modes for applications to choose between. The other online options (**kLookupModeOnlineNoCache**, **kLookupModeOnlineNoCacheRead**, and **kLookupModeOnlineCacheOnly**) are variations of the online mode. These additional online lookup modes give more control over when the SDK is allowed to perform a network connection and how to use the online queries stored locally. The online-query store is used as a performance aid for online queries. If no storage provider is present, no online-query store is utilized.

Setting lookup mode for user code sample (C++):

```
/* Set lookup mode (all queries done with this user will inherit the lookup mode) */
user.Options().LookupMode(kLookupModeLocal);
```

Objective-C:

```
NSError *error = nil;
[[self.gnUser options] lookupModeWithLookupMode:kLookupModeLocal error:&error];
```

Default Lookup Mode

If the application does not set one, the GNSDK sets a default lookup mode—**kLookupModeOnline**—unless the GNSDK license file limits all queries to be local-only, which prevents the SDK from connecting online. When this limit is set in the license file, the lookup mode defaults to **kLookupModeLocal**.

Setting the Lookup Mode for a User or Queries

You can set the lookup mode as a user option or set it separately as a specific query option. Calling the `GnUser.LookupMode` method applies the option to all queries using the user handle. You can also use the `GnMusicId.LookupMode`, `GnMusicIdFile.LookupMode`, or `GnMusicIdStream.LookupMode` methods to override this for specific queries.

User example (C++):

```
GnUser user = GnUser(userStore, clientId, clientIdTag, applicationVersion);
/* Set user to match our desired lookup mode (all queries done with this user will inherit the
lookup mode) */
user.Options().LookupMode(kLookupModeLocal);
```

Query example (C++):

```
/* Perform the query */
music_id.Options().LookupMode(kLookupModeLocal);
GnResponseAlbums response = music_id.FindAlbums(albObject);
```

Query example (Objective-C):

```
GnMusicIdStreamOptions *options = [self.gnMusicIDStream options];
[options lookupMode:kLookupModeLocal error:&error];
```

Using Both Local and Online Lookup Modes

Your application can switch between local and online lookups, as needed.

Identifying Music

The GNSDK supports identifying music in three different modules:

- **Music-ID**—Provides support for identifying music using the following:
 - CD TOC
 - Text search
 - Fingerprints
 - Gracenote identifier

The `GnMusicId` class supports this functionality. See "Identifying Music Using a CD-TOC, Text, or Fingerprints (MusicID)" for more information.

- **Music-ID File**—Provides support for identifying music stored in audio files. The `GnMusicIdFile` class supports this functionality. See "Identifying Audio Files (MusicID-File)" for more information.
- **Music-ID Stream**—Provides support for identifying music that is delivered in real-time as an end-user listens. For example: listening to a song on the radio or playing a song from a media player. You can identify streaming music using audio fingerprints generated from a streaming audio source, typically through a microphone. The `GnMusicIdStream` class provides support for this functionality. On some platforms, the `GnMic` class is available for listening to a device microphone.

For more information, see "Identifying Streaming Music (MusicID-Stream)".

Identifying Music Using a CD-TOC, Text, or Fingerprints (MusicID)

The MusicID module is the GNSDK component that handles recognition of non-streaming music through a CD TOC, audio source, fingerprint or search text. MusicID is implemented using the `GnMusicId` class. The MusicID-File module is the GNSDK component that handles audio file recognition, implemented through the `GnMusicIdFile` class. For information on identifying audio files and using the `GnMusicIdFile` class, see "Identifying Audio Files (MusicID-File)".

MusicID Queries

The `GnMusicId` class provides the following query methods:

- **FindAlbums**—Call this with an album or track identifier such as a CD TOC string, an audio source, a fingerprint, or identifying text (album title, track title, artist name, track artist name or composer name). This method returns a `GnResponseAlbums` object for each matching album.
- **FindMatches**—Call this method with identifying text. The method returns a `GnResponseDataMatches` object for each match, which could identify an album or a contributor.

Notes:

- A `GnMusicId` object's life time is scoped to a single recognition event and your application should create a new one for each event.
- During a recognition event, status events can be received via a delegate object that implements `IGnStatusEvents` (`GnStatusEventsDelegate` in Objective-C).
- A recognition event can be cancelled by the `GnMusicId` `cancel` method or by the "canceller" provided in each events delegate method.

Options for MusicID Queries

The `GnMusicId::Options` class allows you to set the following options:

- **LookupData**—Set `GnLookupData` options to enable what data can be returned, for example, classical data, mood and tempo data, playlist, external IDs, and so on.
- **LookupMode**—Set a lookup option with one of the `GnLookupMode` enums. These include ones for local only, online only, online nocache, and so on.
- **PreferResultLanguage**—Use one of the `GnLanguage` enums to set the preferred language for results.
- **PreferResultExternalId**—Set external ID for results from external provider. External IDs are 3rd party IDs used to cross link this metadata to 3rd party services.
- **PreferResultCoverart**—Specifies preference for results that have cover art associated.
- **ResultSingle**—Specifies whether a response must return only the single best result. Default is `true`.
- **ResultRangeStart**— Specifies result range start value.
- **ResultCount**— Specifies maximum number of returned results.

Identifying Music Using a CD TOC

MusicID-CD is the component of GNSDK that handles recognition of audio CDs and delivery of information including artist, title, and track names. The application provides GNSDK with the TOC from an audio CD and MusicID-CD will identify the CD and provide album and track information.

To identify music using a CD-TOC:

- Instantiate a `GnMusicId` object with your user handle.
- Set a string with your TOC values
- Call the `GnMusicId::FindAlbums` method with your TOC string.
- Process the `GnResponseAlbum` metadata result objects returned.

The code samples below illustrates a simple TOC lookup for local and online systems. The code for the local and online lookups is the same, except for two areas. If you are performing a local lookup, you must initialize the SQLite and Local Lookup libraries, in addition to the other GNSDK libraries:

C++

```
gnsdk_cstr_t toc= "150 14112 25007 41402 54705 69572 87335 98945 112902 131902 144055 157985 176900 189260 203342";
GnMusicId music_id(user);

music_id.Options().LookupData(kLookupDataContent);
GnResponseAlbums response = music_id.FindAlbums(toc);
```

Java and Android Java

```
GnMusicId GnMusicId = new GnMusicId(user);
String toc = "150 14112 25007 41402 54705 69572 87335 98945 112902 131902 144055 157985 176900
189260 203342";
GnResponseAlbums responseAlbums = GnMusicId.findAlbums(toc);
```

Objective-C

Under construction

C# and Windows Phone C#

```
string toc = "150 14112 25007 41402 54705 69572 87335 98945 112902 131902 144055 157985 176900
189260 203342";
try
{
    using (GnStatusEventsDelegate midEvents = new MusicIdEvents())
    {
        GnMusicId GnMusicId = new GnMusicId(user, midEvents);
        GnResponseAlbums gnResponse = GnMusicId.FindAlbums(toc);
    }
}
```

Identifying Music Using Text

Using the GNSDK's MusicID module, your application can identify music using a lookup based on text strings. Besides user-inputted text, text strings can be extracted from an audio track's file path name and from text data embedded within a file, such as mp3 tags. You can provide the following types of input strings:

- Album title
- Track title
- Album artist
- Track artist
- Track composer

Text-based lookup attempts to match these attributes with known albums, artists, and composers. The text lookup first tries to match an album. If that is not possible, it next tries to match an artist. If that does not succeed, a composer match is tried. Adding as many input strings as possible to the lookup improves the results.

Text-based lookup returns "best-fit" objects, which means that depending on what your input text matched, you might get back album matches or contributor matches.

Identifying music using text is done using the `GnMusicId` class that has numerous methods for finding albums, tracks, and matches

To identify music using text:

1. Code an events handler object callbacks for status events (optional).
2. Instantiate a `GnMusicId` object with your User object and events handler object. Note that the events handler is optional.
3. Call the `GnMusicId::FindAlbums` method with your text search string(s).
4. Process metadata results returned

C++


```
/* Set the input text as album title, artist name, track title and perform the query */
GnResponseAlbums response = music_id.FindAlbums("Supernatural", "Africa Bamba", "Santana", GNSDK_
NULL, GNSDK_NULL);
```

Android Java and Java

```
GnMusicId musicId = new GnMusicId( gnUser, new StatusEvents() );
GnResponseAlbums result = musicId.findAlbums( album, track, artist, null, null );
```

Objective-C

```
musicId = [[GnMusicId alloc] initWithGnUser: self.gnUser statusEventsDelegate: self];

[self.cancellableObjects addObject: musicId];

[[musicId options] lookupData:kLookupDataContent bEnable:YES error:&error];

self.queryBeginTimeInterval = [[NSDate date] timeIntervalSince1970];

[self enableOrDisableControls:NO];

dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_BACKGROUND, 0), ^{
    NSError *textSearchOperationError = nil;
    GnResponseAlbums *responseAlbums = [musicId findAlbumsWithAlbumTitle: albumTitle
                                                                    trackTitle: trackTitle
                                                                    albumArtistName: artistName
                                                                    trackArtistName: artistName
                                                                    composerName: nil
                                                                    error: &textSearchOperationError];

    });
}
```

C# and Windows Phone C#

```
/* Set the input text as album title, artist name, track title and perform the query */
GnResponseAlbums gnResponse = musicID.FindAlbums("Supernatural", "Africa Bamba", "Santana");
```

Identifying Music Using Fingerprints

You can identify music using an audio fingerprint. An audio fingerprint is data that uniquely identifies an audio track based on the audio waveform. You can use MusicID or MusicID-File to identify music using an audio fingerprint. The online Gracenote Media Service uses audio fingerprints to match the audio from a client application to the Gracenote Music Database. For more information, see *Fingerprint-Based Recognition*.

PCM Audio Format Recommendations

GNSDK fingerprinting supports the following PCM audio formats:

- **Sample Sizes**—16-bit
- **Channels**—1 or 2 (mono or stereo)
- **Sample Rates**—11025 Hz, 16000 Hz, 22050 Hz, 24000 Hz, 32000 Hz, 44100 Hz, 48000 Hz

Applications should use the highest quality audio possible to ensure the best results. Lower quality audio will result in less accurate fingerprint matches. Gracenote recommends at least 16-bit, stereo, 22050 Hz.



Do not resample or downsample audio to target these frequencies. Send the best quality audio that you have available.

MusicID Fingerprinting

The MusicID fingerprinting APIs give your application the ability to provide audio data as an identification mechanism. Note that if you want to do recognition using fingerprints and metadata together, and possibly have many files to do at once, then MusicID-File fingerprinting is probably the better solution. See *"Identifying Audio Files (MusicID-File)"*

There are four `GnMusicId` fingerprinting methods:

- **FingerprintFromSource**—Generates a fingerprint from a provided audio source. **Gracenote recommends using this**, as it encapsulates the below three calls (and additionally required code) into one.
- **FingerprintBegin**—Initialize fingerprint generation.
- **FingerprintWrite**—Provides uncompressed audio data for fingerprint generation. You can call this after `FingerprintBegin` to generate a native Gracenote Fingerprint Extraction (GNFPX) or Cantamatrix (CMX) fingerprint.
- **FingerprintEnd**—Finalizes fingerprint generation.

Identifying music using MusicID fingerprinting examples:

C++

Under construction

Android Java

Under construction

Objective-C

Under construction

Windows Phone C# code sample

Under construction

C#

```
/*-----  
* SetFingerprintBeginWriteEnd  
*/  
private static void  
SetFingerprintBeginWriteEnd(GnMusicId GnMusicId)  
{  
    bool complete = false;  
  
    FileInfo file = new FileInfo(@"..\..\data\05-Hummingbird-sample.wav");  
  
    using (BinaryReader b = new BinaryReader(File.Open(file.FullName, FileMode.Open,  
FileAccess.Read)))  
    {  
        b.BaseStream.Position = 0;  
    }  
}
```

```

        /* skip the wave header (first 44 bytes). we know the format of our sample files*/
        b.BaseStream.Seek(44, SeekOrigin.Begin);

        byte[] audioData = b.ReadBytes(2048);

        GnMusicId.FingerprintBegin(GnFingerprintType.kFingerprintTypeGNFPX, 44100, 16, 2);

        while (audioData.Length > 0)
        {
            complete = GnMusicId.FingerprintWrite(audioData, (uint)audioData.Length);
            if (true == complete)
                break;
            else
                audioData = b.ReadBytes(2048);
        }

        GnMusicId.FingerprintEnd();

        if (false == complete)
        {
            /* Fingerprinter doesn't have enough data to generate a fingerprint.
            Note that the sample data does include one track that is too short to
            fingerprint. */
            Console.WriteLine("\nWarning: input file does contain enough data to generate a
            fingerprint :\" + file.FullName);
        }
    }
}

/*-----
 * MusicidFingerprintAlbum
 */
private static void
MusicidFingerprintAlbum(GnUser user)
{
    Console.WriteLine("\n*****Sample MID-Stream Query*****");

    try
    {
        GnMusicId GnMusicId = new GnMusicId(user);

        /* Set the input fingerprint */
        SetFingerprintBeginWriteEnd(GnMusicId);

        /* Perform the search */
        GnResponseAlbums response = GnMusicId.FindAlbums(GnMusicId.FingerprintDataGet(),
        GnFingerprintType.kFingerprintTypeGNFPX);

        DisplayFindAlbumResultsByFingerprint(response);
    }
    catch (GnException e)
    {
    }
}

```

Java

Under construction

Best Practices for MusicID Text Searches

When performing a MusicID text search, you can provide values for artist name, album title and track title fields. The more fields you provide, the better and more accurate the result. Therefore, always provide as many fields as possible in your queries. Responses depend on whether the queries are performed using a local database or online:

- Local matches return either an album response (with or without matched track data), or an artist response if an album match was not available.
- Online matches always return album responses, even if the only field provided is an artist name.

In general, an album response can include the following metadata:

- Album
 - Genre
 - Album Image
 - Primary Artist
 - Era
 - Origin
 - Artist Type
 - Artist Image
 - Matched Track (only available if query includes a track title as input)
 - Genre
 - Mood
 - Tempo

The accuracy of a match depends on the input fields you provide and whether the query is performed online or on a local database. The following table shows which of the returned metadata you should display for a given set of inputs.

Album Title	Contributor Name	Track Title	Usable Data from Online Search	Usable Data from Local Search
--------------------	-------------------------	--------------------	---------------------------------------	--------------------------------------

Yes	Yes	Yes	Use all information: <ul style="list-style-type: none"> • Album • Album's Primary Artist • Matched Track if available • Matched Track's Artist if available 	Same as online
Yes	Yes	No	Use all available information: <ul style="list-style-type: none"> • Album • Album's Primary Artist 	Same as online
Yes	No	Yes	Use all available information: <ul style="list-style-type: none"> • Album • Album's Primary Artist • Matched Track if available • Matched Track's Artist if available 	Same as online
Yes	No	No	Do not send this query.	Do not send this query.
No	Yes	Yes	Use all available information: <ul style="list-style-type: none"> • Album • Album's Primary Artist • Matched Track if available • Matched Track's Artist if available 	Same as online
No	Yes	No	Do not use album level info. Use only: <ul style="list-style-type: none"> • Album's Primary Artist 	<ul style="list-style-type: none"> • Use all available information. • Local will only return Artist metadata.
No	No	Yes	Do not send this query.	Do not send this query.

Identifying Audio Files (MusicID-File)

The MusicID-File module can perform recognition using individual audio files or leverage collections of files to provide advanced recognition. When an application provides decoded audio and text data for each file to the library, MusicID-File identifies each file and, if requested, identifies groups of files as albums.

Music-ID File provides three mechanisms for identification:

1. **TrackID**—TrackID identifies the best album(s) for a single track. It returns results for an individual track independent of any other tracks submitted for processing at the same time.
2. **AlbumID**—AlbumID identifies the best album(s) for a group of tracks. Use AlbumID when identifying submitted files as a group is important. For example, if all the submitted tracks were originally recorded on different albums, but exist together on a greatest hits album, then that will be the first match returned.
3. **LibraryID**—LibraryID identifies the best album(s) for a large collection of tracks. Besides metadata and other submitted tracks, LibraryID also takes into account a number of factors, such as location on device, when returning results.

For more information about the differences between TrackID, AlbumID, and LibraryID see the [MusicID-File Overview](#)

MusicID-File is implemented with the `GnMusicIdFile` class.

To identify audio from a file:

1. Code an `IGnMusicIdFileEvents` delegate class (`GnMusicIdFileEventsDelegate` in Objective-C) containing callbacks for results, events, and identification (metadata, fingerprint, and so on.).
2. Instantiate a `GnMusicIdFile` object with your User object and events delegate object.
3. Call the `GnMusicIdFile`'s `FileInfos` method to get a `GnMusicIdFileInfoManager` object.
4. For each file you want to identify, instantiate a `GnMusicIdFileInfo` object for it using the `GnMusicIdFileInfoManager` object's `Add` method.
5. For each instantiated `GnMusicIdFileInfo` object, set the file's path with the object's `FileName` method and assign it an identifier to correlate it with returned results.

Call one of the `GnMusicIdFile` query methods.

- 6.
7. Handle metadata results in an events delegate callback.

Implementing an Events Delegate

To receive `GnMusicIdFile` notifications for results, events, and identification (metadata, fingerprint, and so on.), your application needs to implement the `IGnMusicIdFileEvents` event delegate

(GnMusicIdFileEventsDelegate in Objective-C) provided upon GnMusicIdFile object construction. This events delegate can contain callbacks for the following:

- Results handling
- Status event handling
- Other event handling
- Fingerprinting for identification
- Metadata for identification



Please note that these callbacks are optional, but you will want to code a results handling callback at a minimum, in order to get results. In addition, fingerprint and metadata identification can be done automatically when you create an audio file object (see next section) for each file you want to identify.

Android Java

```
/**
 * GNSDK MusicID-File event delegate
 */
private class MusicIDFileEvents extends IGnMusicIdFileEvents {

    HashMap<String, String> gnStatus_to_displayStatus;

    public MusicIDFileEvents() {
        gnStatus_to_displayStatus = new HashMap<String, String>();
        gnStatus_to_displayStatus.put("kMusicIdFileCallbackStatusProcessingBegin", "Begin processing file");
        gnStatus_to_displayStatus.put("kMusicIdFileCallbackStatusFileInfoQuery", "Querying file info");
        gnStatus_to_displayStatus.put("kMusicIdFileCallbackStatusProcessingComplete", "Identification complete");
    }

    // ...other delegate events
}
```

Objective-C

```
#pragma mark - MusicIDFileEventsDelegate Methods

-(void) musicIdFileAlbumResult: (GnResponseAlbums*)albumResult currentAlbum: (NSUInteger)currentAlbum totalAlbums: (NSUInteger)totalAlbums cancellableDelegate: (id <GnCancellableDelegate>)canceller
{
    [self.cancellableObjects removeObject: canceller];

    if (self.cancellableObjects.count==0)
    {
        self.cancelOperationsButton.enabled = NO;
    }

    [self processAlbumResponseAndUpdateResultsTable:albumResult];
}

if (self.cancellableObjects.count==0)
{
```

```

        self.cancelOperationsButton.enabled = NO;
    }

    [self enableOrDisableControls:YES];
    [self processAlbumResponseAndUpdateResultsTable:result];
}

// ...other delegate events

```

Windows Phone C#

```

#region IGnMusicIdFileEvents

    void IGnMusicIdFileEvents.GatherFingerprint(GnMusicIdFileInfo fileinfo, uint current_file,
    uint total_files, IGnCancellable canceller)
    {
        return;
    }

    void IGnMusicIdFileEvents.GatherMetadata(GnMusicIdFileInfo fileinfo, uint current_file, uint
    total_files, IGnCancellable canceller)
    {
        return;
    }

    void IGnMusicIdFileEvents.MusicIdFileComplete(GnError musicidfile_complete_error)
    {
        List<AlphaKeyGroup<RespAlbum>> DataSource = AlphaKeyGroup<RespAlbum>.CreateGroups
(respAlbList_,
        System.Threading.Thread.CurrentThread.CurrentUICulture,
        (RespAlbum s) => { return s.Title; }, true);

        Deployment.Current.Dispatcher.BeginInvoke(() =>
        {
            TBStatus.Text = "Status : MusicIDFile Completed Successfully";
            ToggleUIBtnsVisibility(true);
            LLRespAlbum.ItemsSource = DataSource;
        });
    }

    // ...other delegate events

```

C++

```

/*
 * Callback delegate classes
 */

/* Callback delegate called when performing MusicID-File operation */
class MusicIDFileEvents : public IGnMusicIdFileEvents
{
public:
    virtual void
    StatusEvent(GnStatus status,
                gnsdk_uint32_t percent_complete,
                gnsdk_size_t bytes_total_sent,
                gnsdk_size_t bytes_total_received,
                IGnCancellable& canceller)
    {

```



```

std::cout << "status (";

switch (status)
{
case gnsdk_status_unknown:
    std::cout <<"Unknown ";
    break;

case gnsdk_status_begin:
    std::cout <<"Begin ";
    break;

case gnsdk_status_connecting:
    std::cout <<"Connecting ";
    break;

case gnsdk_status_sending:
    std::cout <<"Sending ";
    break;

case gnsdk_status_receiving:
    std::cout <<"Receiving ";
    break;

case gnsdk_status_disconnected:
    std::cout <<"Disconnected ";
    break;

case gnsdk_status_complete:
    std::cout <<"Complete ";
    break;

default:
    break;
}

std::cout << ")", % complete ("
                                << percent_complete
                                << ")", sent ("
                                << bytes_total_sent
                                << ")", received ("
                                << bytes_total_received
                                << ")"
                                << std::endl;

    GNSDK_UNUSED(canceller);
}
//... more delegate events
};

```

C#

Under construction

Java

```
//=====
```

```

// Callback delegate classes
//

// Callback delegate called when performing MusicID-File operation
class MusicIDFileEvents implements IGnMusicIdFileEvents {

    @Override
    public void
        musicIdFileStatusEvent( GnMusicIdFileInfo fileInfo, GnMusicIdFileCallbackStatus status, long
currentFile, long totalFiles, IGnCancellable canceller ) {

        }

    @Override
    public void
        musicIdFileAlbumResult( GnResponseAlbums album_result, long current_album, long total_albums,
IGnCancellable canceller ) {
        System.out.println( "\n*Album " + current_album + " of " + total_albums + "*" );

        try {
            displayResult(album_result);
        } catch ( GnException gnException ) {
            System.out.println("GnException \t" + gnException.getMessage());
        }
    }

    @Override
    public void
        musicIdFileMatchResult( GnResponseDataMatches matches_result, long current_match, long total_
matches, IGnCancellable canceller ) {
        System.out.println( "\n*Match " + current_match + " of " + total_matches + "*" );
    }

    @Override
    public void
        musicIdFileResultNotFound( GnMusicIdFileInfo fileInfo, long currentFile, long totalFiles,
IGnCancellable canceller ) {

    }

    @Override
    public void
        musicIdFileComplete( GnError completeError ) {

    }

    @Override
    public void
        gatherFingerprint( GnMusicIdFileInfo fileInfo, long currentFile, long totalFiles, IGnCancellable
canceller ) {

        boolean complete = false;

        try {

            File audioFile = new File( fileInfo.fileName() );

            if (audioFile.exists()) {

                FileInputStream audioFileInputStream = null;

```

```

        DataInputStream audioDataInputStream = null;

        audioFileInputStream = new FileInputStream(audioFile);

        // skip the wave header (first 44 bytes). the format of the sample files is known
        // but please be aware that many wav file headers are larger than 44 bytes!
        audioFileInputStream.skip(44);

        // initialize the fingerprinter
        // Note: The sample files are non-standard 11025 Hz 16-bit mono to save on file size
        fileInfo.fingerprintBegin(11025, 16, 1);

        audioDataInputStream = new DataInputStream(audioFileInputStream);

        byte[] audioBuffer = new byte[BUFFER_READ_SIZE];
        int readSize = 0;
        do {

            // read data, check for -1 to see if we are at end of file
            readSize = audioDataInputStream.read( audioBuffer );
            if ( readSize == -1 ) {
                break;
            }

            complete = fileInfo.fingerprintWrite( audioBuffer, readSize );

            // does the fingerprinter have enough audio?
            if (complete) {
                break;
            }

        }
        while ( (readSize > 0) && (complete == false) );

        audioDataInputStream.close();

        fileInfo.fingerprintEnd();

        if (!complete){
            // Fingerprinter doesn't have enough data to generate a fingerprint.
            // Note that the sample data does include one track that is too short to
            System.out.println("Warning: input file does not contain enough data to g
fingerprint:\n" + audioFile.getPath());
        }

    }

    } catch ( GnException gnException ) {
        System.out.println("GnException \t" + gnException.getMessage());
    } catch ( IOException e ){
        System.out.println( "Exception reading audio file" + e.getMessage() );
    }

}

@Override
public void
gatherMetadata( GnMusicIdFileInfo fileInfo, long currentFile, long totalFiles, IGnCancellable
canceller ) {

    try {

```

```

        // A typical use for this callback is to read file tags (ID3, etc) for the basic
        // metadata of the track. To keep the sample code simple, we went with .wav files
        // and hardcoded in metadata for just one of the sample tracks, index 5 from
        // sampleAudioFile. So, if this isn't the correct sample track, return.
        if ( fileInfo.identifier().equals( '5' ) == false ) {
            return;
        }

        fileInfo.albumArtist( "kardinal offishall" );
        fileInfo.albumTitle ( "quest for fire" );
        fileInfo.trackTitle ( "intro" );

    } catch ( GnException gnException ) {
        System.out.println("GnException \t" + gnException.getMessage());
    }
}

@Override
public void statusEvent(GnStatus status, long percentComplete, long bytesTotalSent, long
bytesTotalReceived, IGnCancellable canceller) {
    // override to receive status events for queries to Gracenote service
}

};

```

Adding Audio Files for Identification

To add audio files for identification:

1. Call the `GnMusicIdFile::FileInfos` method to get a `GnMusicIdFileInfoManager` object.
2. For each file you want to identify, instantiate a `GnMusicIdFileInfo` object for it using the `GnMusicIdFileInfoManager` object's `Add` method.

Each audio file must be added with a unique identifier string that your application can use to correlate results in callbacks with a specific file. Audio files can be added as a `GnAudioFile` (only available on some platforms) instance.

3. Set the file's path with the `GnMusicIdFileInfo::FileName` method.

`GnMusicIdFileInfo` objects are used to contain the metadata that will be used in identification and will also contain results after a query has completed. MusicID-File matches each `GnMusicIdFileInfo` object to a track within an album.



Adding audio files as a `GnAudioFile` instance (only available on some platforms) allows `GnMusicIdFile` to automatically extract metadata available in audio tags (artist, album, track, track number, and so on,) and fingerprint the raw audio data, saving your application the need to do this in event delegate methods.

Setting Audio File Identification

To aid in identification, your application can call `GnMusicIdFileInfo` methods to get/set CDDDB IDs, fingerprint, path and filename, `FileInfo` identifier (set when `FileInfo` created), media ID (from Gracenote), source filename (from parsing) and application, Media Unique ID (MUI), Tag ID (aka Product ID), TOC offsets, track artist/number/title, and TUI (Title Unique Identifier).

Android Java

```
@Override
public void gatherMetadata(GnMusicIdFileInfo fileInfo, long currentFile, long totalFiles,
IGnCancellable cancellable) {

    MediaMetadataRetriever mmr = new MediaMetadataRetriever();
    try {
        mmr.setDataSource(fileInfo.fileName());
        fileInfo.albumTitle(mmr.extractMetadata(MediaMetadataRetriever.METADATA_KEY_ALBUM));
        fileInfo.albumArtist(mmr.extractMetadata(MediaMetadataRetriever.METADATA_KEY_
ALBUMARTIST));
        fileInfo.trackTitle(mmr.extractMetadata(MediaMetadataRetriever.METADATA_KEY_TITLE));
        fileInfo.trackArtist(mmr.extractMetadata(MediaMetadataRetriever.METADATA_KEY_ARTIST));
        try {
            long trackNumber = Long.parseLong(mmr.extractMetadata
(MediaMetadataRetriever.METADATA_KEY_CD_TRACK_NUMBER));
            fileInfo.trackNumber(trackNumber);
        } catch (NumberFormatException e) {}

        } catch (IllegalArgumentException e1) {
            Log.e(appString, "illegal argument to MediaMetadataRetriever.setDataSource");
        } catch (GnException e1) {
            Log.e(appString, "error retrieving filename from fileInfo");
        }
    }
}
```

Objective-C

```
-(void) gatherMetadata: (GnMusicIdFileInfo*) fileInfo
        currentFile: (NSUInteger) currentFile
        totalFiles: (NSUInteger) totalFiles
cancellableDelegate: (id <GnCancellableDelegate>) canceller
{
    NSError *error = nil;
    NSString* filePath = [fileInfo fileName:&error];

    if (error)
    {
        NSLog(@"Error while retrieving filename %@", [error localizedDescription]);
    }
    else
    {
        AVAsset *asset = [AVAsset assetWithURL:[NSURL fileURLWithPath:filePath]];
        if (asset)
        {
            NSArray *metadataArray = [asset metadataForFormat:AVMetadataFormatID3Metadata];

            for(AVMetadataItem* item in metadataArray)
            {
                NSLog(@"AVMetadataItem Key = %@ Value = %@",item.key, item.value );

                if([item commonKey] isEqualToString:@"title")
            }
        }
    }
}
```

```

        {
            [fileInfo trackTitleWithValue:(NSString*) [item value] error:nil];
        }
        else if([[item commonKey] isEqualToString:@"albumName"])
        {
            [fileInfo albumTitleWithValue:(NSString*) [item value] error:nil];
        }
        else if([[item commonKey] isEqualToString:@"artist"])
        {
            [fileInfo trackArtistWithValue:(NSString*) [item value] error:nil];
        }
    }
}
}
}
}

```

Windows Phone C#

```

void IGnMusicIdFileEvents.GatherMetadata(GnMusicIdFileInfo fileInfo, uint current_file, uint total_files, IGnCancellable canceller)
{
    return;
}

```

C++

```

GnMusicIdFileInfo fileInfo;
fileInfo = midf.FileInfos().Add(fileIdent);
fileInfo.FileName(filePath);
fileInfo.AlbumArtist( "kardinal offishall" );
fileInfo.AlbumTitle ( "quest for fire" );
fileInfo.TrackTitle ( "intro" );

```

C#

Under construction

Java

Under construction

MusicID-File Fingerprinting

The MusicID-File fingerprinting APIs give your application the ability to provide audio data as an identification mechanism. This enables MusicID-File to perform identification based on the audio itself, as opposed to performing identification using only the associated metadata. Use the MusicID-File fingerprinting APIs during an events delegate callback.

There are four `GnMusicIdFileInfo` fingerprinting methods:

- **FingerprintFromSource**—Generates a fingerprint from a provided audio source. **Gracenote recommends using this**, as it encapsulates the below three calls (and additionally required code) into one.
- **FingerprintBegin**—Initialize fingerprint generation.

- **FingerprintWrite**—Provides uncompressed audio data for fingerprint generation.
- **FingerprintEnd**—Finalizes fingerprint generation.

Android Java

```
@Override
public void gatherFingerprint(GnMusicIdFileInfo fileInfo, long currentFile, long totalFiles,
IGnCancellable cancelable){
    try {
        fileInfo.fingerprintFromSource( new GnAudioFile( new File(fileInfo.fileName()) ) );
    } catch (GnException e) {
        Log.e(appString, "error in fingerprinting file: " + e.getErrorAPI() + ", " +
e.getErrorModule() + ", " + e.getErrorDescription());
    }
}
```

Objective-C

```
[musicIDFileInfo fingerprintFromSource:(id <GnAudioSourceDelegate> )gnAudioFile error:&error];
```

Windows Phone C#

Under construction

C++

```
file = fileInfo.FileName();

std::ifstream audioFile (file, std::ios::in | std::ios::binary);
if ( audioFile.is_open() )
{
    /* skip the wave header (first 44 bytes). the format of the sample files is known,
    * but please be aware that many wav file headers are larger than 44 bytes!
    */
    audioFile.seekg(44);
    if ( audioFile.good() )
    {
        /* initialize the fingerprinter
        * Note: The sample files are non-standard 11025 Hz 16-bit mono to save on file size
        */
        fileInfo.FingerprintBegin(11025, 16, 1);

        do
        {
            audioFile.read(pcmAudio, 2048);
            complete = fileInfo.FingerprintWrite((gnsdk_byte_t*)pcmAudio,
                                                (gnsdk_size_t)audioFile.gcount()
                                                );

            /* does the fingerprinter have enough audio? */
            if (GNSDK_TRUE == complete)
            {
                break;
            }
        }
        while ( audioFile.good() );

        if (GNSDK_TRUE != complete)
        {
            /* Fingerprinter doesn't have enough data to generate a fingerprint.
            Note that the sample data does include one track that is too short to fingerprint. */

```

```

        std::cout << "Warning: input file does contain enough data to generate a fingerprint:\n"
<< file << "\n";
        fileInfo.FingerprintEnd();
    }
}
else
{
    std::cout << "\n\nError: Failed to skip wav file header: " << file << "\n\n";
}
}
}

```

C#

```

/*-----
 * GatherFingerprint
 */
public override void
GatherFingerprint(GnMusicIdFileInfo fileInfo, uint currentFile, uint totalFiles, IGnCancellable
canceller)
{
    byte[]    audioData  = new byte[2048];
    bool      complete   = false;
    int       numRead    = 0;
    FileStream fileStream = null;

    try
    {
        string filename = fileInfo.FileName;
        if (filename.Contains('\'))
            fileStream = new FileStream(filename, FileMode.Open, FileAccess.Read);
        else
            fileStream = new FileStream(folderPath + filename, FileMode.Open, FileAccess.Read);

        /* check file for existence */
        if (fileStream == null || !fileStream.CanRead)
        {
            Console.WriteLine("\n\nError: Failed to open input file: " + filename);
        }
        else
        {
            /* skip the wave header (first 44 bytes). we know the format of our sample files, but
please
            be aware that many wav file headers are larger then 44 bytes! */
            if (44 != fileStream.Seek(44, SeekOrigin.Begin))
            {
                Console.WriteLine("\n\nError: Failed to seek past header: %s\n", filename);
            }
            else
            {
                /* initialize the fingerprinter
                Note: Our sample files are non-standard 11025 Hz 16-bit mono to save on file size
*/
                fileInfo.FingerprintBegin(11025, 16, 1);

                numRead = fileStream.Read(audioData, 0, 2048);
                while ((numRead) > 0)
                {
                    /* write audio to the fingerprinter */
                    complete = fileInfo.FingerprintWrite(audioData, Convert.ToUInt32(numRead));

                    /* does the fingerprinter have enough audio? */

```



```
        if (complete)
        {
            break;
        }

        numRead = fileStream.Read(audioData, 0, 2048);
    }
    fileStream.Close();

    /* signal that we are done */
    fileInfo.FingerprintEnd();
    Debug.WriteLine("Fingerprint: " + fileInfo.Fingerprint + " File: " +
fileInfo.FileName);
    }
}

if (!complete)
{
    /* Fingerprinter doesn't have enough data to generate a fingerprint.
    Note that the sample data does include one track that is too short to fingerprint. */
    Console.WriteLine("Warning: input file does contain enough data to generate a
fingerprint:\n" + filename);
}

}

catch (FileNotFoundException e)
{
    Console.WriteLine("FileNotFoundException " + e.Message);
}

catch (IOException e)
{
    Console.WriteLine("IOException " + e.Message);
}

finally
{
    try
    {
        fileStream.Close();
    }
    catch (IOException e)
    {
        Console.WriteLine("IOException " + e.Message);
    }
}
}
```

Java

Under construction

Setting Options for MusicID-File Queries

To set an option for your MusicID-File query, instantiate a `GnMusicIdFileOptions` object using the `GnMusicIdFile::Options` method and call its methods. For example, you can set an option for local lookup. By default, a lookup is handled online, but many applications will want to start with a local query first then, if no match is returned, fall back to an online query.

MusicID-File Query Options:

- **LookupData**—Set `GnLookupData` options to enable what data can be returned, for example, classical data, mood and tempo data, playlist, external IDs, and so on.
- **LookupMode**—Set a lookup option with one of the `GnLookupMode` enums. These include ones for local only, online only, online nocache, and so on.
- **BatchSize**—In `LibraryID`, you can set the batch size to control how many files are processed at a time. The higher the size, the more memory will be used. The lower the size, the less memory will be used and the faster results will be returned.
- **ThreadPriority**—Use one of the `GnThreadPriority` enums to set thread priority, for example, default, low, normal, high, and so on.
- **OnlineProcessing**—Enable (`true`) or disable (`false`) online processing.
- **PreferResultLanguage**—Use one of the `GnLanguage` enums to set the preferred language for results.
- **PreferResultExternalId**—Set external ID for results from external provider. External IDs are 3rd party IDs used to cross link this metadata to 3rd party services.

Making a MusicID-File Query

`GnMusicIdFile` provides the following query methods:

- **DoTrackId**—Perform a Track ID query.
- **DoTrackIdAsync**—Perform an asynchronous Track ID query.
- **DoAlbumId**—Perform an Album ID query.
- **DoAlbumIdAsync**—Perform an asynchronous Album ID query.
- **DoLibraryId**—Perform a Library ID query.
- **DoLibraryIdAsync**—Perform an asynchronous Library ID query.



Note that GNSDK processing is always done asynchronously and returns results via callbacks. With the blocking functions, your app waits until processing has completed before continuing.

Options When Making Query Call

When you make a `GnMusicIdFile` query method call, you can set the following options at the time of the call (as opposed to setting options with `GnMusicIdFileOptions` object—see above):

- **Return matches** - Have MusicID-File return all results found for each given `GnMusicIdFileInfo`
- **Return albums** - Only album matches are returned (default)
- **Return all** - Have MusicID-File return all results found for each given `GnMusicIdFileInfo`
- **Return single** - Have MusicID-File return the single best result for each given `GnMusicIdFileInfo` (default)

C++

```
/* Launch AlbumID */
midf.DoAlbumId(kQueryReturnSingle, kResponseAlbums );
```

Objective-C

```
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
    NSError *error = nil;
```

```
[musicIDFileInfo fingerprintFromSource:(id <GnAudioSourceDelegate> )gnAudioFile error:&error];
error = nil;
[gnMusicIDFile doAlbumId:kQueryReturnSingle responseType:kResponseAlbums error:&error];
});
```

Identifying Streaming Music (MusicID-Stream)

The functionality for identifying streaming audio is contained in the **Music-ID Stream** module. This module contains the `GnMusicIdStream` class that is designed to identify raw audio received in a continuous stream. You should instantiate one instance of this class for each audio stream you are using.

`GnMusicIdStream` can be started and stopped as the audio stream starts and stops. There is no need to destroy and recreate a `GnMusicIdStream` instance due to breaks in the audio stream.

Using this class, your application primarily needs to provide two things:

1. Code an `IGnMusicIdStreamEvents` events delegate object (`GnMusicIdStreamEventsDelegate` in Objective-C) that receives callbacks for results, status messages, and other events.
2. Code a class that implements the `IGnAudioSource` (`GnAudioSource` in Objective-C) interface.



For some platforms (for example, iOS and Android), Gracenote provides the `GnMic` helper class that implements the `IGnAudioSource` interface. If available, your application can use this class to process streaming audio from a device microphone.

Notes:

- At any point, your application can stop audio processing. When stopped, automatic data fetching ceases or, if audio data is being provided manually, attempts to write data for processing will fail. Internally, `GnMusicIdStream` clears and releases all buffers and audio processing modules. Audio processing can be restarted at any time.
- Identification spawns a thread that completes asynchronously. However, methods for both synchronous and asynchronous identification are provided. If the synchronous method is called, the identification is still performed asynchronously and results delivered via a delegate implementing `IGnMusicIdStreamEvents`, and the method does not return until the identification is complete. If a request is pending, any new ID requests are ignored.
- Audio is identified using either a local database or the Gracenote online service. The default is to attempt a local identification first before going online. Local matches are only possible if `GnLookupLocalStream` is enabled and a MusicID-Stream fingerprint bundle ingested.
- Internally, `GnMusicIdStream` pulls data from the audio source interface in a loop, so you may want to start automatic audio processing in a background thread to avoid stalling the main thread.
- At any point, your application can request an identification of buffered audio. The identification process spawns a thread and completes asynchronously. Use the method `IdentifyAsync` to identify music.
- You can call `IdentifyCancel` to stop an identification operation. Note that cancelling does not

cease audio processing and your application can continue requesting identifications.

- You can instantiate a `GnMusicIdStream` object with a locale. Locales are a convenient way to group locale-dependent metadata specific to a region (such as Europe) and language that should be returned from the Gracenote service. See "Loading a Locale" for more information.

To identify streaming audio from an audio source (`IGnAudioSource` implementation):

1. Code a `IGnMusicIdStreamEvents` delegate class (`GnMusicIdStreamEventsDelegate` in Objective-C) to handle results, statuses, and other events.
2. Instantiate a `GnMusicIdStream` object with your User object, a `GnMusicIdStreamPreset` enum, and a `IGnMusicIdStreamEvents` events delegate object. This establishes a `MusicID-Stream` audio channel. The `GnMusicIdStreamPreset` enum can be for either 'microphone' or 'radio' (for example, speaker).
3. Instantiate a `IGnAudioSource` object representing the audio source you wish to identify. On some platforms (iOS, Android, Windows Phone), Gracenote provides the `GnMic` class, which is a `IGnAudioSource` implementation for the device microphone.
4. Call the `IGnAudioSource` instance's `sourceInit` method to initialize your audio source.
5. Call the `GnMusicIdStream` instance's `AudioProcessStart` method with your `IGnAudioSource` object. This starts the retrieval and processing of audio.



Note that, as an alternative to the above two steps, you could call the `GnMusicIdStream` instance's `audioProcess` method with PCM string data that you have captured through any means.

6. To identify audio, call `GnMusicIdStream.IdentifyAlbumAsync`. Results are delivered asynchronously as audio is received.
7. Handle results and statuses in your event delegate callbacks.
8. If your app goes into the background, call `audioProcessStop`; this releases resources and relinquishes control of the audio source. If it returns to the foreground, call `audioProcessStart` again.

Examples

Android Java code sample

```
// Set up for continuous listening from the microphone
// - Create microphone, this can live for lifetime of app
// - Create GnMusicIdStream instance, this can live for lifetime of app
// - Set options
GnMicrophone = new GnMic(); // Init with default samples per second (8000), bits per sample (16)
and num channels (1)
GnMusicIdStream = new GnMusicIdStream( gnUser, GnMusicIdStreamPreset.kPresetMicrophone, new
MusicIDStreamEvents() );
GnMusicIdStream.options().lookupData(GnLookupData.kLookupDataContent, true);
GnMusicIdStream.options().lookupData(GnLookupData.kLookupDataSonicData, true);
GnMusicIdStream.options().resultSingle( true );

// Start audio processing with GnMic, GnMusicIdStream pulls data from GnMic internally
```

```

gnMicrophone.sourceInit(); // Initialize microphone
gnMusicIdStream.audioProcessStart( gnMicrophone );

// Perform identification
gnMusicIdStream.identifyAlbumAsync();

// Handle results in callback

```

Objective-C code sample

```

mMic = [[GnMic alloc] initWithSampleRate:44100 bitsPerChannel:8 numberOfChannels:2];
mMusicAudio = [[GnMusicIdStream alloc] initWithGnUser:self.gnUser preset:kPresetMicrophone
locale:self.locale musicIdStreamEventsDelegate: self];
-(void)start
{
    NSError *error = nil;

    if(mMic & (0 == [mMic sourceInit]))
    {
        if (mMusicAudio)
        {
            [mMusicAudio audioProcessStartWithSamplesPerSecond:[mMic samplesPerSecond] bitsPerSample:
[mMic sampleSizeInBits] numberOfChannels:[mMic numberOfChannels] error:&error];
        }
    }
    [self performSelectorInBackground:@selector(processAudio) withObject:nil];
}

-(void)processAudio
{
    NSError *error = nil;
    NSMutableData *data = [[NSMutableData alloc] initWithLength:1024];
    while (true)
    {
        NSInteger length = [mMic getData:data];
        if(data)
        {
            [mMusicAudio audioProcess:data error:&error];
        }
    }
}

//Call this when user request identification
[mMusicAudio identifyAlbumAsync:&error];

// Handle results in callback

```

Windows Phone C# code sample

```

gnMic = new GnMic();
gnMic.SourceInit();
App.gnMusicIDStream_ = new GnMusicIdStream(App.gnMusicUser_, nMusicIdStreamPreset.kPresetMicrophone,
this);
App.gnMusicIDStream_.Options.ResultSingle(true);
App.gnMusicIDStream_.Options.LookupData(GnLookupData.kLookupDataContent, true);

if (gnMic.SampleSizeInBits() == 16)
{
    App.gnMusicIDStream_.AudioProcessStart(gnMic.SamplesPerSecond(), 16, gnMic.NumberOfChannels());
}

```

```

}
else
{
    App.gnMusicIDStream_.AudioProcessStart(gnMic.SamplesPerSecond(), 8, gnMic.NumberOfChannels());
}
//...
bytesRead = gnMic.GetData(audioData, (uint)audioData.Length);
App.gnMusicIDStream_.AudioProcess(audioData, bytesRead);

// Call this when user requests identification
App.gnMusicIDStream_.IdentifyAlbumAsync();

//Handle results in callback

```

Setting Options for Streaming Audio Queries

You can use `GnMusicIdStreamOptions` methods to set options for streaming audio queries. For example, you can set an option for local lookup. By default, a lookup is done online, but many applications will want to start with a local query first then, if no match is found, go online.

GnMusicIdStreamOptions Query Option Methods:

- **LookupData**—Set `GnLookupData` options to enable what data can be returned, for example, classical data, mood and tempo data, playlist, external IDs, and so on.
- **LookupMode**—Set a lookup option with one of the `GnLookupMode` enums. These include ones for local only, online only, online nocache, and so on.
- **NetworkInterface**—Set a specific network interface to use with this object's connections.
- **PreferResultLanguage**—Use one of the `GnLanguage` enums to set the preferred language for results.
- **PreferResultExternalId**—Specifies preference for results with external IDs. External IDs are 3rd party IDs used to cross link this metadata to 3rd party services.
- **PreferResultCoverart**—Specifies preference for results with cover art.
- **ResultSingle**—Specifies whether a response must return only the single best result. Default is `true`.
- **ResultRangeStart**—Specifies the result range start value. This must be less than or equal to the total number of results. If greater than the total number, no results are returned.
- **ResultCount**—Specifies the result range count value.

Accessing Classical Music Metadata

GNSDK supports classical music metadata through `GnAudioWork` objects. This topic describes how to access this metadata. For an overview, see [Classical Music Metadata](#).

The following code sample, shows how to navigate a classical music audio work and iterate through the returned track. For the full example, see the `musicid_gdo_navigation` sample application. Also see `GnResponseAlbums` in the [OO Data Model](#) and navigate to `GnAlbum > GnAudioWork`.

Example Classical Music Output

Below is an example of rendered XML output for a classical album.

```

<ALBUM>
  <PACKAGE_LANG ID="1" LANG="eng">English</PACKAGE_LANG>

```

```

<ARTIST>
  <NAME_OFFICIAL>
    <DISPLAY>Various Artists</DISPLAY>
  </NAME_OFFICIAL>
  <CONTRIBUTOR>
    <NAME_OFFICIAL>
      <DISPLAY>Various Artists</DISPLAY>
    </NAME_OFFICIAL>
  </CONTRIBUTOR>
</ARTIST>
<TITLE_OFFICIAL>
  <DISPLAY>Grieg: Piano Concerto, Peer Gynt Suites #1 & 2</DISPLAY>
</TITLE_OFFICIAL>
<YEAR>1989</YEAR>
<GENRE_LEVEL1>Classical</GENRE_LEVEL1>
<GENRE_LEVEL2>Romantic Era</GENRE_LEVEL2>
<LABEL>LaserLight</LABEL>
<TOTAL_IN_SET>1</TOTAL_IN_SET>
<DISC_IN_SET>1</DISC_IN_SET>
<TRACK_COUNT>3</TRACK_COUNT>
<COMPILATION>Y</COMPILATION>
<TRACK_ORD="1">
  <TRACK_NUM>1</TRACK_NUM>
  <ARTIST>
    <NAME_OFFICIAL>
      <DISPLAY>János Sándor: Budapest Philharmonic Orchestra</DISPLAY>
    </NAME_OFFICIAL>
    <CONTRIBUTOR>
      <NAME_OFFICIAL>
        <DISPLAY>János Sándor: Budapest Philharmonic Orchestra</DISPLAY>
      </NAME_OFFICIAL>
      <ORIGIN_LEVEL1>Eastern Europe</ORIGIN_LEVEL1>
      <ORIGIN_LEVEL2>Hungary</ORIGIN_LEVEL2>
      <ORIGIN_LEVEL3>Hungary</ORIGIN_LEVEL3>
      <ORIGIN_LEVEL4>Hungary</ORIGIN_LEVEL4>
      <ERA_LEVEL1>2000's</ERA_LEVEL1>
      <ERA_LEVEL2>2000's</ERA_LEVEL2>
      <ERA_LEVEL3>2000's</ERA_LEVEL3>
      <TYPE_LEVEL1>Mixed</TYPE_LEVEL1>
      <TYPE_LEVEL2>Mixed Group</TYPE_LEVEL2>
    </CONTRIBUTOR>
  </ARTIST>
  <TITLE_OFFICIAL>
    <DISPLAY>Grieg: Piano Concerto In A Minor, Op. 16</DISPLAY>
  </TITLE_OFFICIAL>
  <GENRE_LEVEL1>Classical</GENRE_LEVEL1>
  <GENRE_LEVEL2>Other Classical</GENRE_LEVEL2>
</TRACK>
<TRACK_ORD="2">
  <TRACK_NUM>2</TRACK_NUM>
  <ARTIST>
    <NAME_OFFICIAL>
      <DISPLAY>Yuri Ahronovitch: Vienna Symphony Orchestra</DISPLAY>
    </NAME_OFFICIAL>
    <CONTRIBUTOR>
      <NAME_OFFICIAL>
        <DISPLAY>Yuri Ahronovitch: Vienna Symphony Orchestra</DISPLAY>
      </NAME_OFFICIAL>
      <ORIGIN_LEVEL1>Western Europe</ORIGIN_LEVEL1>
      <ORIGIN_LEVEL2>Austria</ORIGIN_LEVEL2>
    </CONTRIBUTOR>
  </ARTIST>

```

```

    <ORIGIN_LEVEL3>Vienna</ORIGIN_LEVEL3>
    <ORIGIN_LEVEL4>Vienna</ORIGIN_LEVEL4>
    <ERA_LEVEL1>1990&apos;s</ERA_LEVEL1>
    <ERA_LEVEL2>1990&apos;s</ERA_LEVEL2>
    <ERA_LEVEL3>1990&apos;s</ERA_LEVEL3>
    <TYPE_LEVEL1>Mixed</TYPE_LEVEL1>
    <TYPE_LEVEL2>Mixed Group</TYPE_LEVEL2>
  </CONTRIBUTOR>
</ARTIST>
<TITLE_OFFICIAL>
  <DISPLAY>Grieg: Peer Gynt Suite #1, Op. 46</DISPLAY>
</TITLE_OFFICIAL>
<GENRE_LEVEL1>Classical</GENRE_LEVEL1>
<GENRE_LEVEL2>Other Classical</GENRE_LEVEL2>
</TRACK>
<TRACK ORD="3">
  <TRACK_NUM>3</TRACK_NUM>
  <ARTIST>
    <NAME_OFFICIAL>
      <DISPLAY>Daniel Gerard; Peter Wohlert: Berlin Radio Symphony Orchestra</DISPLAY>
    </NAME_OFFICIAL>
    <CONTRIBUTOR>
      <NAME_OFFICIAL>
        <DISPLAY>Daniel Gerard; Peter Wohlert: Berlin Radio Symphony Orchestra</DISPLAY>
      </NAME_OFFICIAL>
      <ORIGIN_LEVEL1>Western Europe</ORIGIN_LEVEL1>
      <ORIGIN_LEVEL2>Germany</ORIGIN_LEVEL2>
      <ORIGIN_LEVEL3>Berlin</ORIGIN_LEVEL3>
      <ORIGIN_LEVEL4>Berlin</ORIGIN_LEVEL4>
      <ERA_LEVEL1>1990&apos;s</ERA_LEVEL1>
      <ERA_LEVEL2>Early 90&apos;s</ERA_LEVEL2>
      <ERA_LEVEL3>Early 90&apos;s</ERA_LEVEL3>
      <TYPE_LEVEL1>Mixed</TYPE_LEVEL1>
      <TYPE_LEVEL2>Mixed Group</TYPE_LEVEL2>
    </CONTRIBUTOR>
  </ARTIST>
  <TITLE_OFFICIAL>
    <DISPLAY>Grieg: Peer Gynt Suite #2, Op. 55</DISPLAY>
  </TITLE_OFFICIAL>
  <GENRE_LEVEL1>Classical</GENRE_LEVEL1>
  <GENRE_LEVEL2>Other Classical</GENRE_LEVEL2>
</TRACK>
</ALBUM>

```

Processing Returned Metadata Results

Processing returned metadata results is pretty straight-forward—objects returned can be traversed and accessed like any other objects. However, there are three things about returned results you need to be aware of:

1. **Needs decision**—A result could require an additional decision from an application or end user.
2. **Full or partial results**—A result object could contain full or partial metadata

3. **Locale-dependent data**—Some metadata requires that a locale be loaded.

Needs Decision

Top-level response classes—`GnResponseAlbums`, `GnResponseTracks`, `GnResponseContributors`, `GnResponseDataMatches`—contain a "needs decision" flag. In all cases, Gracenote returns high-confidence results based on the identification criteria; however, even high-confidence results could require additional decisions from an application or end user. For example, all multiple match responses require the application (or end user) make a decision about which match to use and process to retrieve its metadata. Therefore, the GNSDK flags every multiple match response as "needs decision".

A single match response can also need a decision. Though it may be a good candidate, Gracenote could determine that it is not quite perfect.

In summary, responses that require a decision are:

- Every multiple match response
- A single match response that Gracenote determines needs a decision from the application or end user, based on the quality of the match and/or the mechanism used to identify the match (such as text, TOC, fingerprints, and so on).

Example needs decision check (C++):

```
GnResponseAlbums response = music_id.FindAlbums(albObject);
needs_decision = response.NeedsDecision();
if(needs_decision)
{
    /* Get user selection. Note that is not an SDK method */
    user_pick = doMatchSelection(response);
}
```

Objective-C:

```
/* Perform the query */
GnResponseAlbums *response = [musicId findAlbumsWithGnDataObject:dataAlbum error:&error];

if ( [[response albums] allObjects].count != 0)
{
    needsDecision = [response needsDecision];

    /* See if selection of one of the albums needs to happen */
    if (needsDecision)
    {
        // Get User selection. Note that is not an SDK method
        choiceOrdinal = [self doMatchSelection:response];
    }
    // ...
}
```

Full and Partial Metadata Results

A query response can return 0-n matches. As indicated with a `fullResult` flag, a match can contain either full or partial metadata results. A partial result is a subset of the full metadata available, but enough to

perform additional processing. One common use case is to present a subset of the partial results (for example, album titles) to the end user to make a selection. Once a selection is made, you can then do a secondary query, using the partial object as a parameter, to get the full results (if desired).

Example followup query (C++):

```
/* Get first match */
GnAlbum album = response.Albums().at(0).next();
bool fullResult = album.IsFullResult();

/* If partial result, do a follow-up query to retrieve full result */
if (!fullResult)
{
    /* Do followup query to get full data - set partial album as query input. */
    GnResponseAlbums followup_response = music_id.FindAlbums(album);

    /* Now our first album has full data */
    album = followup_response.Albums().at(0).next();
}
```

Objective-C:

```
GnAlbum *album = [[response albums] allObjects] objectAtIndex:choiceOrdinal];

BOOL fullResult = [album isFullResult];

/* if we only have a partial result, we do a follow-up query to retrieve the full album */
if (!fullResult)
{
    /* do followup query to get full object. Setting the partial album as the query input. */
    GnResponseAlbums *followupResponse = [musicId findAlbumsWithGnDataObject:dataAlbum error:&error];

    /* now our first album is the desired result with full data */
    album = [[followupResponse albums] nextObject];

    // ...
}
```



Note that, in many cases, the data contained in a partial result is more than enough for most users and applications. For a list of values returned in a partial result, see the Data Model in the GNSDK API Reference.

Locale-Dependent Data

GNSDK provides locales as a convenient way to group locale-dependent metadata specific to a region (such as Europe) and language that should be returned from the Gracenote service. A locale is defined by a group (such as Music), a language, a region and a descriptor (indicating level of metadata detail), which are identifiers to a specific set of lists in the Gracenote Service.

There are a number of metadata fields that require having a locale loaded. For more information, see "Loading a Locale".

Accessing Enriched Content

To access enriched metadata content, such as cover art and artist images, you can purchase additional metadata entitlements and use the content URLs returned from queries in your application. Enriched content URLs can be returned in every result object (`GnAlbum`, `GnTrack`, etc).

To access enriched content:

- Purchase additional entitlements for enriched content
- Enable the query option for retrieving enriched content
- For each match object returned, iterate through its `GnContent` objects
- For each `GnContent` object, iterate through its `GnAsset` objects
- For each `GnAsset` object, get its content URL and use that for accessing the Gracenote service

Setting the Query Option for Enriched Content

To get enriched content returned from your queries, you need to enable the query option for this. You can do this using the `LookupData` method and the `kLookupDataContent` enum.

C++ example:

```
/* Enable retrieval of enriched content */
musicid.Options().LookupData(kLookupDataContent, true);
```

Objective-C example:

```
GnMusicIdStreamOptions *options = [self.gnMusicIDStream options];
[options lookupData:kLookupDataContent enable:YES error:&musicIDStreamError];
```

Processing Enriched Content

Enriched content is returned in `GnContent` objects. As indicated with these `GnContentType` enums, the following types of content can be returned:

- `kContentTypeImageCover`—Cover art
- `kContentTypeImageArtist`—Artist image
- `kContentTypeImageVideo`—Video image
- `kContentTypeBiography`—Artist biography
- `kContentTypeReview`—Review

Each `GnContent` object can contain one or more elements of these types as a `GnAsset` object.

Note that not all enriched content can be retrieved from every metadata object. Different objects have different types of enriched content available. The following classes contain these `GnContent` objects:

- `GnTrack` - Review
- `GnAlbum` - CoverArt
- `GnAlbum` - Review
- `GnContributor` - Image
- `GnContributor` - Biography

The `GnContributor` class also has a `BiographyVideo` field as a `String`.

Each `GnAsset` object contains the following fields:

- **Dimension**—Asset dimension
- **Bytes**—Size of content asset in bytes
- **Size**—Pixel image size as defined with a `GnImageSize` enum, for example, `kImageSize110` (110x110)
- **url**—URL for retrieval of asset from Gracenote service.

Retrieving a Content Asset

You can use the `GnAssetFetch` class to access an asset from local storage, the Gracenote service, or any Internet location with a `GnAsset` URL field and retrieve its content as raw byte data. The asset is retrieved with a `GnAssetFetch` object constructor:

```
GnAssetFetch(GnUser user, String url, IGnStatusEvents pEventHandler)
```

This call takes your `User` object, the URL as a string, and a `IGnStatusEvents` delegate callback object for handling operation statuses

When the operation completes, the asset data is stored in the `GnAssetFetch` object's `data` field as a byte array.



Whether your asset is retrieved locally or online depends on how your lookup mode is set. See [Setting Local and Online Lookup Modes](#) for more information. Note that retrieving an asset online could result in some network delay depending on asset size.

Retrieving and Parsing Enriched Content Code Samples:

Android Java

```
/**
 * Helpers to load and set cover art image in the application display
 */
private void loadAndDisplayCoverArt( GnAlbum album, ImageView imageView ){
    Thread runThread = new Thread( new CoverArtLoaderRunnable( album, imageView ) );
    runThread.start();
}

class CoverArtLoaderRunnable implements Runnable {
    GnAlbum album;
    ImageView imageView;
    CoverArtLoaderRunnable( GnAlbum album, ImageView imageView){
        this.album = album;
        this.imageView = imageView;
    }

    @Override
    public void run() {
        String coverArtUrl = album.content(GnContentType.
            kContentTypeImageCover).
            asset(GnImageSize.kImageSizeSmall).url();
        Drawable coverArt = null;
        if (coverArtUrl != null && !coverArtUrl.isEmpty()) {
            URL url;
```

```

        try {
            url = new URL("http://" + coverArtUrl);
            InputStream input = new
            BufferedInputStream(url.openStream());
            coverArt =
                Drawable.createFromStream(input, "src");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    if (coverArt != null) {
        setCoverArt(coverArt, imageView);
    } else {
        setCoverArt(getResources().getDrawable(R.drawable.no_cover_art), imageView);
    }
}
}
private void setCoverArt( Drawable coverArt, ImageView coverArtImage ){
    activity.runOnUiThread(new SetCoverArtRunnable(coverArt, coverArtImage));
}
class SetCoverArtRunnable implements Runnable {
    Drawable coverArt;
    ImageView coverArtImage;
    SetCoverArtRunnable( Drawable locCoverArt, ImageView locCoverArtImage) {
        coverArt = locCoverArt;
        coverArtImage = locCoverArtImage;
    }
    @Override
    public void run() {
        coverArtImage.setImageDrawable(coverArt);
    }
}
}

```

Objective-C

```

for(GnAlbum* album in albums)
{
    /* Get CoverArt */
    GnContent *coverArtContent = [album content:kContentTypeImageCover];
    GnAsset *coverArtAsset = [coverArtContent asset:kImageSizeSmall];
    NSString *URLString = [NSString stringWithFormat:@"http://%@", [coverArtAsset url]];
    GnContent *artistImageContent = nil; // [album content:kContentTypeImageArtist];
    GnAsset *artistImageAsset = [artistImageContent asset:kImageSizeSmall];
    NSString *artistImageURLString = [NSString stringWithFormat:@"http://%@", [artistImageAsset
url]];
    GnContent *artistBiographyContent = [album content:kContentTypeBiography];
    NSString *artistBiographyURLString = [NSString stringWithFormat:@"http://%@",
[[[artistBiographyContent assets] nextObject] url]];
    GnContent *albumReviewContent = [album content:kContentTypeReview];
    NSString *albumReviewURLString = [NSString stringWithFormat:@"http://%@", [[[albumReviewContent
assets] nextObject] url]];
}

```

Windows Phone C#

```

public RespAlbum(gnsdk_cppcx.GnAlbum gnAlbum, bool bTakeMachedTrack)
{
    this.Title = gnAlbum.Title.Display;
    if (false == bTakeMachedTrack)
    {
        if(0 != gnAlbum.Tracks.Count())
    }
}

```

```

        this.TrackTitle = gnAlbum.Tracks.ElementAt(0).Title.Display;
    else
        this.TrackTitle = "";
    }
    else
    {
        if(0 != gnAlbum.TracksMatched.Count())
            this.TrackTitle = gnAlbum.TracksMatched.ElementAt(0).Title.Display;
        else
            this.TrackTitle = "";
    }
    this.ArtistName = gnAlbum.Artist.Name.Display;
    this.Genre = gnAlbum.Genre(gnsdk_cppcx.GnDataLevel.kDataLevel_1);
    this.ImageUrl = "http://" + gnAlbum.Content(gnsdk_
cppcx.GnContentType.kContentTypeImageCover).Asset(gnsdk_cppcx.GnImageSize.kImageSizeSmall).Url;
    if ("http://" == ImageUrl)
    {
        this.ImageUrl = "/Assets/emptyImage.png";
    }
}

```

C++

```

static void fetchCoverArtfromUrl(GnUser& user, gnsdk_cstr_t url)
{
    static gnsdk_uint32_t count = 1;
    gnsdk_char_t buf[MAX_BUF_SIZE] = {0};
    gnsdk_byte_t *img = NULL;
    gnsdk_size_t size = 0;
    GnAssetFetch gnAssetFetch(user, url);

    snprintf(buf, MAX_BUF_SIZE, "cover_art_%d.jpg", count);
    std::ofstream coverart(buf, std::ios::out | std::ios::binary);

    if (coverart.is_open())
    {
        img = gnAssetFetch.Data();
        size = gnAssetFetch.Size();
        if (img && size)
        {
            std::cout<<<"Url:"<<<url<<<"as "<<<buf<<<std::endl<<<std::endl;
            coverart.write((const char *)img, size);
        }
        count++;
    }
}

static void navigateAlbumResponse(GnUser& user, GnAlbum& album)
{
    content_iterator Iter = album.Contents().begin();

    for (; Iter != album.Contents().end(); ++Iter)
    {
        GnContent content = *Iter;
        asset_iterator aIter = content.Assets().begin();
        for (; aIter != content.Assets().end(); ++aIter)
        {
            GnAsset asset = *aIter;
            if (kContentTypeImageCover == content.ContentType() )
            {

```

```

        fetchCoverArtfromUrl(user, asset.Url());
    }
}
}
}

```

C#

```

/*-----
 * FetchCoverArt
 */
private static void
FetchCoverArt(GnUser user)
{
    Console.WriteLine("\n*****Sample Link Album Query*****");

    using (LookupStatusEvents linkStatusEvents = new LookupStatusEvents())
    {
        // The below serialized GDO was an 1-track album result from another GNSDK query.
        string serializedGdo =
"WEcxAbwXl+DYDXSI3nZZ/L9ntBr8EhRjYAYzNEwlFNyCWkbGGLvyitwgmBccgJtgIM/dkcbDgrOqBMIQJZMmnyvsjCkx10ppXc68
ZcgU0SgLelyjfo1Tt7Ix/cn32BvcbeuPkAk0WwwReVdcSLu08cYxAGcGQrEE+4s2H75HwxFG28r/yb2QX71pR";

        // Typically, the GDO passed in to a Link query will come from the output of a GNSDK query.
        // For an example of how to perform a query and get a GDO please refer to the documentation
        // or other sample applications.

        GnMusicId gnMusicID = new GnMusicId(user);
        gnMusicID.Options().LookupData(GnLookupData.kLookupDataContent, true);

        GnResponseAlbums responseAlbums = gnMusicID.FindAlbums(GnDataObject.Deserialize(serializedGdo));
        GnAlbum gnAlbum = responseAlbums.Albums.First<GnAlbum>();

        GnLink link = new GnLink(gnAlbum, user, null);
        if (link != null)
        {
            // Cover Art
            GnLinkContent coverArt = link.CoverArt(GnImageSize.kImageSize170,
GnImagePreferenceType.kImagePreferenceSmallest);
            byte[] coverData = coverArt.DataBuffer;

            // save coverart to a file.
            fetchImage(coverData.Length.ToString(), "cover art");
            if (coverData != null)
                File.WriteAllBytes("cover.jpeg", coverData);

            // Artist Image
            GnLinkContent imageArtist = link.ArtistImage(GnImageSize.kImageSize170,
GnImagePreferenceType.kImagePreferenceSmallest);
            byte[] artistData = imageArtist.DataBuffer;

            // save artist image to a file.
            fetchImage(artistData.Length.ToString(), "artist");
            if (artistData != null)
                File.WriteAllBytes("artist.jpeg", artistData);
        }
    }
}

```

Java code sample

```

//=====
// fetchImage
//      Display file size
//
void
fetchImage( GnLink link, GnLinkContentType contentType, String imageTypeStr )
{
    GnImagePreferenceType imgPreference = GnImagePreferenceType.kImagePreferenceSmallest;
    GnImageSize            imageSize      = GnImageSize.kImageSize170;
    String                 fileName       = null;
    GnLinkContent           linkContent    = null;

    // Perform the image fetch
    try
    {
        // For image to be fetched, it must exist in the size specified and you must be entitled to fetch
        images.

        switch ( contentType )
        {
        case kLinkContentCoverArt:
            linkContent = link.coverArt( imageSize, imgPreference );
            fileName     = "cover.jpg";
            break;

        case kLinkContentImageArtist:
            linkContent = link.artistImage( imageSize, imgPreference );
            fileName     = "artist.jpg";
            break;
        }

        // Do something with the image, e.g. display, save, etc. Here we just print the size.
        long dataSize = linkContent.dataSize();
        System.out.printf( "\nRETRIEVED: %s: %d byte JPEG\n", imageTypeStr, dataSize );

        // get image data
        byte[] imageData = new byte[(int) dataSize];
        linkContent.dataBuffer(imageData);

        // Save image to file.
        DataOutputStream os = new DataOutputStream( new FileOutputStream(fileName) );
        os.write( imageData );
        os.close();
    }
    catch ( GnException e )
    {
        System.out.println( e.errorAPI() + "\t" + e.errorCode() + "\t" + e.errorDescription() );
    }
    catch ( IOException e ) {
        System.out.println( e.getMessage() );
    }
}

```


Discovery Features

Generating a Playlist

You can easily integrate the Playlist SDK into your media management application. Note that your application should already have identifiers for all its media and its own metadata database. The Playlist module allows your application to create Playlists—sets of related media—from larger collections.

Collection Summaries, that you create, are the basis for generating playlists. Collection Summaries contain attribute data about a given set of media. For Playlist operations—create, populate, store, delete, and so on.—use the `GnPlaylistCollection` class.

To generate a Playlist:

1. Create a Collection Summary
2. Populate the Collection Summary with media objects, most likely returned from Gracenote queries
3. (Optional) Store the Collection Summary
4. (Optional) Load the stored Collection Summary into memory in preparation for Playlist results generation
5. Generate a Playlist from a Collection Summary using either the More Like This functionality or with a Playlist Description Language (PDL) statement (see the Playlist PDL Specification).
6. Access and display Playlist results.

Creating a Collection Summary

To create a Collection Summary, your application needs to instantiate a `GnPlaylistCollection` object:

C++

```
playlist::GnPlaylistCollection myCollection;  
myCollection= playlist::GnPlaylistCollection("MyCollection");
```

C#

```
GnPlaylistCollection playlistCollection;  
playlistCollection = GnPlaylist.CollectionCreate("sample_collection");
```

Java

```
GnPlaylistCollection myCollection = new GnPlaylistCollection();
```

This call creates a new, empty Collection Summary. The next step is to populate it with media items that you can use to generate Playlists.



Note: Each new Collection Summary that you create must have a unique name. Although it is possible to create more than one Collection Summary with the same name, if these Collection Summaries are then saved to local storage, one Collection will override the other. To avoid this, ensure that Collection Summary names are unique.

Populating a Collection Summary

To build a Collection Summary, your application needs to provide data for each media item you want it to contain. To add an item, and provide data, use the `GnPlaylistCollection.Add` method. This API takes a media identifier (any application-determined unique string) and an album, track or contributor match object. The match object should come from a recognition event using MusicID, MusicID-File or other GNSDK module.

C++

```
/* Create a unique identifier for every track that is added to the playlist.
   Ideally the ident allows for the identification of which track it is.
   for example path/filename.ext , or an id that can be externally looked up.
*/
ss.str("");
ss << index << "_" << ntrack;

/*
   Add the the Album and Track GDO for the same ident so that we can
   query the Playlist Collection with both track and album level attributes.
*/
std::string result = ss.str();
collection.Add(result.c_str(), album);      /* Add the album*/
collection.Add(result.c_str(), *itr);       /* Add the track*/
```

C#

```
/* Create a unique identifier for every track that is added to the playlist.
   Ideally, the identifier allows for the identification of which track it is.
   for example path/filename.ext , or an id that can be externally looked up.
*/
string uniqueIdent = countOrdinal + "_" + trackOrdinal; playlistCollection.Add(uniqueIdent, album);

/*
   Add the the Album and Track GDO for the same identifier so we can
   query the Playlist Collection with both track and album level attributes.
*/
playlistCollection.Add(uniqueIdent, album);
playlistCollection.Add(uniqueIdent, track);
```

Java

```
String uniqueIdent = "";
uniqueIdent = String.valueOf(index).concat("_").concat(String.valueOf(ntrack));

/*
 * Add the the Album and Track GDO for the same identifier so we can
 * query the Playlist Collection with both track and album level attributes.
 */
```

```
gnPlaylistCollection.add(uniqueIdent, gnAlbum);  
gnPlaylistCollection.add(uniqueIdent, gnTrack);
```

Retrieving Playlist Attributes in Queries

When creating a MusicID or MusicID-File query to populate a playlist, you must set the following query options to ensure that the appropriate Playlist attributes are returned (depending on the type of query):

- `GnLookupData.kLookupDataSonicData`
- `GnLookupData.kLookupDataPlaylist`

C++

```
musicId.Options().LookupData(kLookupDataSonicData, true);  
musicId.Options().LookupData(kLookupDataPlaylist, true);
```

How Playlist Gathers Data

When given an album match object, Playlist extracts necessary album data, traverses to the matched track, and extracts necessary track data. Playlist stores this data for the given identifier. If the album object does not contain the matched track, no track data is collected. Playlist also gathers data from both the album and track contributors as detailed below.

When given a track match object, Playlist gathers any necessary data from the track, but it is not able to gather any album-related data (such as album title). Playlist also gathers data from the track contributors as detailed below.

When given a contributor match object (or traversing into one from an album or track), Playlist gathers the necessary data from the contributor. If the contributor is a collaboration, data from both child contributors is gathered as well.

Working with Local Storage

You can store and manage Collection Summaries in local storage with the `GnPlaylistStorage's Store` method, which takes a `GnPlaylistCollection` object. **Prior to doing this, your application should enable a storage solution such as SQLite.**

C++

```
/* Instantiate SQLite module to use as our database engine*/  
GnStorageSqlite storageSqlite;  
  
/* This module defaults to use the current folder when initialized,  
** but we set it manually here to demonstrate the option.  
*/  
storageSqlite.Options().StorageLocation(".");  
  
/* Initialize Storage for storing Playlist Collections */  
playlist::GnPlaylistStorage plStorage;  
plStorage.Store(myCollection);
```

C#

```
/* Initialize Storage for storing Playlist Collections */  
playlist::GnPlaylistStorage plStorage;
```

```
gnStorage.StorageFolder = "../../../sample_db";
playlist.StoreCollection(playlistCollection);
```

Java

```
// Initialize Storage for storing Playlist Collections
GnStorageSqlite storage = new GnStorageSqlite();
GnPlaylistStorage plStorage = GnPlaylist.collectionStorage();
plStorage.store(myCollection);
```

Other `GnPlaylistStorage` methods include ones for:

- Setting a storage location specifically for playlist collections
- Removing a collection from storage
- Loading a collection from storage
- Getting the number of collections stored
- Compacting storage

Generating a Playlist Using More Like This

To streamline your Playlist implementation, you can use the `GnPlaylistCollection`'s `GenerateMoreLikeThis` method, which uses the "More Like This" algorithm to obtain results, eliminating the need to use Playlist Definition Language (PDL) statements.

You can use the `GnPlaylistMoreLikeThisOptions` class to set the following options when generating a More Like This Playlist. Please note that these options are not serialized or stored.

Option Method	Description
<code>MaxTracks</code>	Maximum number of tracks returned in a 'More Like This' playlist. Must evaluate to a number greater than 0.
<code>MaxPerArtist</code>	Maximum number of tracks per artist returned in a 'More Like This' playlist. Must evaluate to a number greater than 0.
<code>MaxPerAlbum</code>	Maximum number of results per album returned. Must evaluate to a number greater than 0.
<code>RandomSeed</code>	Randomization seed value used in calculating a More Like This playlist. Must evaluate to a number greater than 0. To re-create a playlist, use the same number - different numbers create different playlists. If "0", using a random seed is disabled.

C++

```
/* Change the possible result set to be a maximum of 30 tracks. */
collection.MoreLikeThisOptions().MaxTracks(30);

/* Change the max per artist to be 20 */
collection.MoreLikeThisOptions().MaxPerArtist(20);

/* Change the max per album to be 5 */
collection.MoreLikeThisOptions().MaxPerAlbum(5);
```

To generate a More Like This playlist, call the `GnPlaylistCollection`'s `GenerateMoreLikeThis` method with a user object handle and a Gracenote data object, such as the object of the track that is currently playing.

C++

```
/* Generating more like this Playlist */
/* Create seed data to generate more like this playlist*/
/*
 * A seed gdo can be any recognized media gdo.
 * In this example, we are using a gdo from a track in the playlist collection summary,
 * randomly selecting the 5th element
 */
playlist::GnPlaylistIdentifier ident      = collection.MediaIdentifiers().at(4).next();
playlist::GnPlaylistMetadata seed_album = collection.Metadata(user, ident);
playlist::GnPlaylistResult resultMoreLikeThis = collection.GenerateMoreLikeThis(user, seed_album,
topColl));
```

C#

```
/*
 * A seed gdo can be any recognized media gdo.
 * In this example we are using the a gdo from a random track in the playlist collection summary
 */
GnPlaylistIdentifier identifier = playlistCollection.MediaIdentifiers.at(3).Current;
GnPlaylistMetadata data = playlistCollection.Metadata(user, identifier);
playlistCollection.MoreLikeThisOptionSet
(GnPlaylistCollection.GenerateMoreLikeThisOption.kMoreLikeThisMaxPerAlbum, 5);
GnPlaylistResult playlistResult = playlistCollection.GenerateMoreLikeThis(user, data);
```

Java

```
GnPlaylistMetadata getSeedData( GnUser gnUser, GnPlaylistCollection collection ) throws GnException
{
    // Create seed data to generate more like this playlist
    // A seed gdo can be any recognized media gdo.
    // In this example we are using the a gdo from a track in the playlist collection summary
    // In this case , randomly selecting the 5th element
    GnPlaylistIdentifier ident      = collection.mediaIdentifiers().at(4).next();
    GnPlaylistMetadata seedAlbum = collection.metadata( gnUser, ident );
    return seedAlbum;
}

//...

// Set options
collection.options().moreLikeThis( GnPlaylistMoreLikeThisOption.kMoreLikeThisMaxTracks, 30 );
collection.options().moreLikeThis( GnPlaylistMoreLikeThisOption.kMoreLikeThisMaxPerArtist, 10 );
collection.options().moreLikeThis( GnPlaylistMoreLikeThisOption.kMoreLikeThisMaxPerAlbum, 5 );

GnPlaylistResult resultCustomMoreLikeThis = collection.generateMoreLikeThis( gnUser, getSeedData(
gnUser, collection ) );
```

Generating a Playlist Using PDL (Playlist Description Language)

The GNSDK Playlist Definition Language (PDL) is a query syntax, similar to Structured Query Language (SQL), that enables flexible custom playlist generation using human-readable text strings. PDL allows developers to dynamically create custom playlists. By storing individual PDL statements, applications can create and manage multiple preset and user playlists for later use.

PDL statements express the playlist definitions an application uses to determine what items are included in resulting playlists. PDL supports logic operators, operator precedence, and in-line arithmetic. PDL is based on Structured Query Language (SQL). This section assumes you understand SQL and can write SQL statements.



Note: Before using PDL statements, carefully consider if the provided More Like This functionality meets your design requirements. More Like This functionality eliminates the need to create and validate PDL statements.

To generate a playlist using PDL, you can use the `GnPlaylistCollections`'s `GeneratePlaylist` method which takes a PDL statement as a string and returns `GnPlaylistResult` objects.

To understand how to create a PDL statement, see the *Playlist PDL Specification* article.

Accessing Playlist Results

When you generate a Playlist from a Collection Summary, using either More Like This or executing a PDL statement, results are returned in a `GnPlaylistResult` object.

C++

```

/*-----
 *  display_playlist_results
 */
void display_playlist_results(GnUser& user, playlist::GnPlaylistCollection& collection,
playlist::GnPlaylistResult& result)
{
    /* Generated playlist count */
    int resultCount = result.Identifiers().count();

    printf("Generated Playlist: %d\n", resultCount);
    playlist::result_iterator itr = result.Identifiers().begin();

    for (; itr != result.Identifiers().end(); ++itr)
    {
        playlist::GnPlaylistMetadata data = collection.Metadata(user, *itr);

        printf("Ident '%s' from Collection '%s':\n"
            "\tGN_AlbumName : %s\n"
            "\tGN_ArtistName : %s\n"
            "\tGN_Era : %s\n"
            "\tGN_Genre : %s\n"
            "\tGN_Origin : %s\n"
            "\tGN_Mood : %s\n"
            "\tGN_Tempo : %s\n",
            (*itr).MediaIdentifier(),
            (*itr).CollectionName(),
            data.AlbumName(),
            data.ArtistName(),
            data.Era(),
            data.Genre(),
            data.Origin(),
            data.Mood(),
            data.Tempo()
        );
    }
}

```

C#

```

private static void
EnumeratePlaylistResults(GnUser user, GnPlaylistCollection playlistCollection, GnPlaylistResult
playlistResult)
{
    GnPlaylistMetadata gdoAttr = null;
    string ident = null;
    string collectionName = null;
    uint countOrdinal = 0;
    uint resultsCount = 0;
    GnPlaylistCollection tempCollection = null;

    resultsCount = playlistResult.Identifiers.count();

    Console.WriteLine("Generated Playlist: " + resultsCount);

    GnPlaylistResultIdentEnumerable playlistResultIdentEnumerable = playlistResult.Identifiers;
    foreach (GnPlaylistIdentifier playlistIdentifier in playlistResultIdentEnumerable)
    {
        collectionName = playlistIdentifier.CollectionName;
        ident = playlistIdentifier.MediaIdentifier;

        Console.Write("    " + ++countOrdinal + ": " + ident + " Collection Name:" +
collectionName);

        /* The following illustrates how to get a collection handle
           from the collection name string in the results enum function call.
           It ensures that Joined collections as well as non joined collections will work with
minimal overhead.
        */
        tempCollection = playlistCollection.JoinSearchByName(collectionName);

        gdoAttr = tempCollection.Metadata(user, playlistIdentifier);

        PlaylistGetAttributeValue(gdoAttr);

    }
}

private static void
PlaylistGetAttributeValue(GnPlaylistMetadata gdoAttr)
{
    /* Album name */
    if (gdoAttr.AlbumName != "")
        Console.WriteLine("\n\t\tGN_AlbumName:" + gdoAttr.AlbumName);

    /* Artist name */
    if (gdoAttr.ArtistName != "")
        Console.WriteLine("\t\tGN_ArtistName:" + gdoAttr.ArtistName);

    /* Artist Type */
    if (gdoAttr.ArtistType != "")
        Console.WriteLine("\t\tGN_ArtistType:" + gdoAttr.ArtistType);

    /*Artist Era */
    if (gdoAttr.Era != "")
        Console.WriteLine("\t\tGN_Era:" + gdoAttr.Era);

    /*Artist Origin */
    if (gdoAttr.Origin != "")

```

```

        Console.WriteLine("\t\tGN_Origin:" + gdoAttr.Origin);

    /* Mood */
    if (gdoAttr.Mood != "")
        Console.WriteLine("\t\tGN_Mood:" + gdoAttr.Mood);

    /*Tempo*/
    if (gdoAttr.Tempo != "")
        Console.WriteLine("\t\tGN_Tempo:" + gdoAttr.Tempo);
}

```

Java

```

private static void enumeratePlaylistResults(
    GnUser user,
    GnPlaylistCollection playlistCollection,
    GnPlaylistResult playlistResult
) throws GnException {
    int countOrdinal = 0;
    GnPlaylistMetadata data = null;
    GnPlaylistCollection tempCollection = null;
    String collectionName = null;
    System.out.println("Generated Playlist: " + playlistResult.identifiers().count());

    GnPlaylistResultIdentIterator gnPlaylistResultIdentIterator = playlistResult.identifiers().begin
();

    /* Iterate through results */
    while (gnPlaylistResultIdentIterator.hasNext()) {
        GnPlaylistIdentifier playlistIdentifier = gnPlaylistResultIdentIterator.next();

        collectionName = playlistIdentifier.collectionName();

        System.out.println("    " + ++countOrdinal + ": " + playlistIdentifier.mediaIdentifier() +
            " Collection Name:" + playlistIdentifier.collectionName());

        /* The following illustrates how to get a collection handle
           from the collection name string in the results enum function call.
           It ensures that Joined collections as well as non joined collections will work with
minimal overhead.
        */
        tempCollection = playlistCollection.joinSearchByName(collectionName);

        data = tempCollection.metadata(user, playlistIdentifier);

        playlistGetAttributeValue(data);
    }
}

private static void playlistGetAttributeValue(GnPlaylistMetadata data) {

    /* Album name */
    if (data.albumName() != "" && data.albumName() != null)
        System.out.println("\t\tGN_AlbumName:" + data.albumName());

    /* Artist name */
    if (data.artistName() != "" && data.artistName() != null)
        System.out.println("\t\tGN_ArtistName:" + data.artistName());

    /* Artist Type */
    if (data.artistType() != "" && data.artistType() != null)

```



```
System.out.println("\t\tGN_ArtistType:" + data.artistType());

/*Artist Era */
if (data.era() != "" && data.era() != null)
    System.out.println("\t\tGN_Era:" + data.era());

/*Artist Origin */
if (data.origin() != "" && data.origin() != null)
    System.out.println("\t\tGN_Origin:" + data.origin());

/* Mood */
if (data.mood() != "" && data.mood() != null)
    System.out.println("\t\tGN_Mood:" + data.mood());

/*Tempo*/
if (data.tempo() != "" && data.tempo() != null)
    System.out.println("\t\tGN_Tempo:" + data.tempo());
}
```

Working with Multiple Collection Summaries

Creating a playlist across multiple collections can be accomplished by using joins. Joins allow you to combine multiple collection summaries at run-time, so that they can be treated as one collection by the playlist generation functions. Joined collections can be used to generate More Like This and PDL-based playlists.

For example, if your application has created a playlist based on one device (collection 1), and another device is plugged into the system (collection 2), you might want to create a playlist based on both of these collections. This can be accomplished using one of the `GnPlaylistCollection` join methods.



Joins are run-time constructs for playlist generation that support seamless identifier enumeration across all contained collections. They do not directly support the addition or removal of data objects, synchronization, or serialization across all collections in a join. To perform any of these operations, you can use the join management functions to access the individual collections and operate on them separately.

To remove a collection from a join, call the `GnPlaylistCollection`'s `JoinRemove` method.

Join Performance and Best Practices

Creating a join is very efficient and has minimal CPU and memory requirements. When collections are joined, GNSDK internally sets references between them, rather than recreating them. Creating, deleting, and recreating joined collections when needed can be an effective and high-performing way to manage collections.

The original handles for the individual collections remain functional, and you can continue to operate on them in tandem with the joined collection, if needed. If you release an original handle for a collection that has been entered into a joined collection, the joined collections will continue to be functional as long as the collection handle representing the join remains valid.

A good practice for managing the joining of collections is to create a new collection handle that represents the join, and then join all existing collections into this handle. This helps remove ambiguity as to which original collection is the parent collection representing the join.

Synchronizing Collection Summaries

Collection summaries must be refreshed whenever items in the user's media collection are modified. For example, if you've created a collection summary based on the media on a particular device, and the media on that device changes, your application must synchronize the Collection Summary.

1. Adding all existing (current and new) unique identifiers, using the `GnPlaylistCollection's SyncProcessAdd` method.
2. Calling the `GnPlaylistCollection's SyncProcessExecute` method to process the current and new identifiers and using the callback function to add or remove identifiers to or from the Collection Summary.

Iterating the Physical Media

The first step in synchronizing is to iterate through the physical media, calling the `GnPlaylistCollection's SyncProcessAdd` method for each media item. For each media item, pass the unique identifier associated with the item to the method. The unique identifiers used must match the identifiers that were used to create the Collection Summary initially.

Processing the Collection

After preparing a Collection Summary for synchronization using the `GnPlaylistCollection's SyncProcessAdd` method, call the `GnPlaylistCollection's SyncProcessExecute` method to synchronize the Collection Summary's data. During processing, the callback function will be called for each difference between the physical media and the collection summary. This means the callback function will be called once for each new media item, and once for each media item that has been deleted from the collection. The callback function should add new and delete old identifiers from the Collection Summary.

Playlist PDL Specification

The GNSDK for Mobile Playlist Definition Language (PDL) is a query syntax, similar to Structured Query Language (SQL), that enables flexible custom playlist generation using human-readable text strings. PDL allows developers to dynamically create custom playlists. By storing individual PDL statements, applications can create and manage multiple preset and user playlists for later use.

PDL statements express the playlist definitions an application uses to determine what items are included in resulting playlists. PDL supports logic operators, operator precedence and in-line arithmetic. PDL is based on Search Query Language (SQL). This section assumes you understand SQL and can write SQL statements.



NOTE: Before implementing PDL statement functionality for your application, carefully consider if the provided More Like This function, `gnsdk_playlist_generate_morelikethis()` meets your design requirements. Using the More Like This function eliminates the need to create and validate PDL statements.

PDL Syntax

This topic discusses PDL keywords, operators, literals, attributes, and functions.

Note: Not all keywords support all operators. Use `gnsdk_playlist_statement_validate()` to check a PDL Statement, which generates an error for invalid operator usage.

Keywords

PDL supports these keywords:

Keyword	Description	Required or Optional	PDL Statement Order
GENERATE PLAYLIST	All PDL statements must begin with either GENERATE PLAYLIST or its abbreviation, GENPL	Required	1
WHERE	Specifies the attributes and threshold criteria used to generate the playlist. If a PDL statement does not include the WHERE keyword, Playlist operates on the entire collection.	Optional	2
ORDER	Specifies the criteria used to order the results' display. If a PDL statement does not include the ORDER keyword, Playlist returns results in random order. Example: Display results in based on a calculated similarity value; tracks having greater similarity values to input criteria display higher in the results. The expression format is: <identifier> <operator> <identifier>	Optional	3
LIMIT	Specifies criteria used to restrict the number of returned results. Also uses the keywords RESULT and PER. Example: Limiting the number of tracks displayed in a playlist to 30 results with a maximum of two tracks per artist. The expression format is: <identifier> <operator> <identifier>	Optional	4

Keyword	Description	Required or Optional	PDL Statement Order
SEED	<p>Specifies input data criteria from one or more ids. Typically, a Seed is the This in a More Like This playlist request.</p> <p>Example: Using a Seed of Norah Jones' track Don't Know Why to generate a playlist of female artists of a similar genre.</p> <p>The expression format is: <identifier> <operator> <identifier></p>	Optional	NA

Example: Keyword Syntax

This PDL statement example shows the syntax for each keyword. In addition to <att_imp>, <expr>, and <score> discussed above, this example shows:

- <math_op> is one of the valid PDL mathematical operators.
- <number> is positive value.
- <attr_name> is a valid attribute, either a Gracenote-delivered attribute or an implementation-specific attribute.

```
GENERATE PLAYLIST
WHERE <att_imp> [<math_op> <score>][ AND|OR <att_imp>]
ORDER <expr>[ <math_op> <expr>]
LIMIT [number RESULT | PER <attr_name>][,number [ RESULT | PER <attr_name>]]
```

Operators

PDL supports these operators:

Operator Type	Available Operators
Comparison	>, >=, <, <=, ==, !=, LIKE LIKE is for fuzzy matching, best used with strings; see PDL Examples
Logical	AND, OR
Mathematical	+, -, *, /

Literals

PDL supports the use of single (') and double (") quotes, as shown:

- Single quotes: 'value one'
- Double quotes: "value two"

- Single quotes surrounded by double quotes: "value three"



You must enclose a literal in quotes or Playlist evaluates it as an attribute.

Attributes

Most attributes require a Gracenote-defined numeric identifier value or GDO (GnDataObject in object-oriented languages) for Seed.

- Identifier: Gracenote-defined numeric identifier value is typically a 5-digit value; for example, the genre identifier 24045 for Rock. These identifiers are maintained in lists in Gracenote Service; download the lists using GNSDK for Mobile Manager's Lists and List Types APIs
- GDO Seed: Use GNSDK Manager's GDO APIs to access XML strings for Seed input.

The following table shows the supported system attributes and their respective required input. The first four attributes are the GOET attributes.



The delivered attributes have the prefix GN_ to denote they are Gracenote standard attributes. You can extend the attribute functionality in your application by implementing custom attributes; however, do not use the prefix GN_.

Name	Attribute	Required Input
Genre Origin Era Artist Type Mood Tempo	GN_Genre GN_Origin GN_Era GN_ArtistType GN_Mood GN_Tempo	Gracenote-defined numeric identifier value GDO Seed XML string
Artist Name Album Name	GN_ArtistName GN_AlbumName	Text string

Functions

PDL supports these functions:

- RAND(max value)
- RAND(max value, seed)

PDL Statements

This topic discusses PDL statements and their components.

Attribute Implementation <att_imp>

A PDL statement is comprised of one or more attribute implementations that contain attributes, operators, and literals. The general statement format is:

```
"GENERATE PLAYLIST WHERE <attribute> <operator> <criteria>"
```

You can write attribute implementations in any order, as shown:

```
GN_ArtistName == "ACDC"
```

or

```
"ACDC" == GN_ArtistName
```

WHERE and ORDER statements can evaluate to a score; for example:

```
"GENERATE PLAYLIST WHERE LIKE SEED > 500"
```

WHERE statements that evaluate to a non-zero score determine what ids are in the playlist results.

ORDER statements that evaluate to a non-zero score determine how ids display in the playlist results.

Expression <expr>

An expression performs a mathematical operation on the score evaluated from a attribute implementation.

```
[<number> <math_op>] <att_imp>
```

For example:

```
3 * (GN_Era LIKE SEED)
```

Score <score>

Scores can range between -1000 and 1000.

For boolean evaluation, True equals 1000 and False equals 0.

Note: For more complex statement scoring, concatenate attribute implementations and add weights to a PDL statement.

Example: PDL Statements

The following PDL example generates results that have a genre similar to and on the same level as the seed input. For example, if the Seed genre is a Level 2: Classic R&B/Soul, the matching results will include similar Level 2 genres, such as Neo-Soul.

```
"GENERATE PLAYLIST WHERE GN_Genre LIKE SEED"
```

This PDL example generates results that span a 20-year period. Matching results will have an era value from the years 1980 to 2000.

```
"GENERATE PLAYLIST WHERE GN_Era >= 1980 AND GN_Era < 2000"
```

This PDL example performs fuzzy matching with Playlist, by using the term LIKE and enclosing a string value in single (') or double (") quotes (or both, if needed). It generates results where the artist name may be a variation of the term ACDC, such as:

- ACDC
- AC/DC
- AC*DC

```
"GENERATE PLAYLIST WHERE (GN_ArtistName LIKE 'ACDC')"
```

The following PDL example generates results where:

- The tempo value must be less than 120 BPM.
- The ordering displays in descending order value, from greatest to least (119, 118, 117, and so on).
- The genre is similar to the Seed input.

```
"GENERATE PLAYLIST WHERE GN_Tempo > 120 ORDER GN_Genre LIKE SEED"
```

Implementing Mood

The Mood library allows applications to generate playlists and user interfaces based on Gracenote Mood descriptors. Mood provides Mood descriptors to the application in a two-dimensional grid that represents varying degrees of moods across each axis. One axis represents energy (calm to energetic) and the other axis represents valence (dark to positive). When the user selects a mood from the grid, the application can provide a playlist of music that corresponds to the selected mood. Additional filtering support is provided for genre, origin, and era music attributes.

For more information on Moodgrid, see the *Moodgrid Overview*

The Moodgrid APIs:

- Encapsulate Gracenote's Mood Editorial Content (mood layout and ids).
- Simplify access to Mood results through x,y coordinates.
- Allow for multiple local and online data sources through Mood Providers.
- Enable pre-filtering of results using genre, origin, and era attributes.
- Support 5x5 or 10x10 MoodGrids.
- Provide the ability to go from a cell of a 5x5 Mood to any of its expanded four Moods in a 10x10 grid.

Implementing Mood in an application involves the following steps:

1. Allocating a `GnMoodgrid` class.
2. Enumerating the data sources using Mood Providers
3. Creating and populating a Mood Presentation
4. Filtering the results, if needed

Prerequisites

Using the Mood APIs requires the following modules:

- GNSDK Manager
- SQLite (for local caching)
- MusicID
- Playlist
- Mood

If you are using MusicID to recognize music, you must enable Playlist data in your query. You must be entitled to use Playlist—if you are not, you will not get an error, but Mood will return no results. Please contact your Gracenote Global Service & Support representative for more information.

Enumerating Data Sources using Mood Providers

GNSDK automatically registers all local and online data sources available to Mood. For example, if you create a playlist collection using the Playlist API, GNSDK automatically registers that playlist as a data source available to Mood. These data sources are referred to as *Providers*. Mood is designed to work with multiple providers. You can iterate through the list of available Providers using the `GnMoodgrid` class' `providers` method. For example, the following call returns a handle to the first Provider on the list (at index 0):

C++

```
moodgrid::GnMoodgrid myMoodgrid;  
moodgrid::GnMoodgridProvider myProvider = *(myMoodgrid.Providers().at(0));
```

C#

```
GnMoodgrid moodgrid = new GnMoodgrid();  
GnMoodgridProvider provider = moodgrid.Providers.at(0).next();
```

Java

```
GnMoodgrid myMoodGrid = new GnMoodgrid();  
GnMoodgridProvider myProvider = myMoodGrid.providers().at(0).next();
```

You can use the `GnMoodGridProvider` object to retrieve the following information

- Name
- Type
- Requires network

Creating and Populating a Mood Presentation

Once you have a `GnMoodgrid` object, you can create and populate a Mood Presentation with Mood data. A Presentation is a `GnMoodgridPresentation` object that represents the Mood, containing the mood name and playlist information associated with each grid cell.

To create a Mood Presentation, use the `GnMoodgrid` class' "create presentation" method, passing in the user handle and the Mood type. The type can be one of the enumerated values in `GnMoodgridPresentationType`: either a 5x5 or 10x10 grid. The method returns a `GnMoodgridPresentation` object:

C++


```
moodgrid::GnMoodgridPresentation myPresentation = myMoodgrid.CreatePresentation(user,
GnMoodgridPresentationType.kMoodgridPresentationType5x5);
```

C#

```
presentation = moodGrid.CreatePresentation(user,
GnMoodgridPresentationType.kMoodgridPresentationType10x10);
```

Java

```
GnMoodgridPresentation myPresentation = myMoodGrid.createPresentation(user,
GnMoodgridPresentationType.kMoodgridPresentationType5x5);
```

Iterating Through a Mood Presentation

Each cell of the Presentation is populated with a mood name and associated playlist. You can iterate through the Presentation to retrieve this information from each cell:

C++

```
/* Create a moodgrid presentation for the specified type */
moodgrid::GnMoodgridPresentation myPresentation = myMoodgrid.CreatePresentation(user, type);

moodgrid::GnMoodgridPresentation::data_iterator itr = myPresentation.Moods().begin();

for (; itr != myPresentation.Moods().end(); ++itr) {

    /* Find the recommendation for the mood */
    moodgrid::GnMoodgridResult result = myPresentation.FindRecommendations(myProvider, *itr);

    printf("\n\n\tX:%d\tY:%d\tMood Name: %s\tMood ID: %s\tCount: %d\n", itr->X, itr->Y,
myPresentation.MoodName(*itr), myPresentation.MoodId(*itr), result.Count());

    moodgrid::GnMoodgridResult::iterator result_itr = result.Identifiers().begin();

    /* Iterate the results for the identifiers */
    for (; result_itr != result.Identifiers().end(); ++result_itr) {
        printf("\n\n\tX:%d\tY:%d", itr->X, itr->Y);

        printf("\nident:\t%s\n", result_itr->MediaIdentifier());
        printf("group:\t%s\n", result_itr->Group());

        playlist::GnPlaylistMetadata data = collection.Metadata(user, result_itr->MediaIdentifier
(), result_itr->Group());

        printf("Album:\t%s\n", data.AlbumName());
        printf("Mood :\t%s\n", data.Mood());
    }
}
```

C#

```
/* Create a moodgrid presentation for the specified type */
presentation = moodGrid.CreatePresentation(user, gnMoodgridPresentationType);

/* Query the presentation type for its dimensions */
GnMoodgridDataPoint dataPoint = moodGrid.Dimensions(gnMoodgridPresentationType);
Console.WriteLine("\n PRINTING MOODGRID " + dataPoint.X + " x " + dataPoint.Y + " GRID ");

/* Enumerate through the moodgrid getting individual data and results */
```

```
GnMoodgridPresentationDataEnumerable moodgridPresentationDataEnumerable = presentation.Moods;

foreach (GnMoodgridDataPoint position in moodgridPresentationDataEnumerable)
{
    uint x = position.X;
    uint y = position.Y;

    /* Get the name for the grid coordinates in the language defined by Locale */
    string name = presentation.MoodName(position);

    /* Get the mood id */
    string id = presentation.MoodId(position);

    /* Find the recommendation for the mood */
    GnMoodgridResult moodgridResult = presentation.FindRecommendations(provider, position);

    /* Count the number of results */
    count = moodgridResult.Count();
    Console.WriteLine("\n\n\tX:" + x + "   Y:" + y + " name: " + name + " count: " + count + " ");

    /* Iterate the results for the ids */
    GnMoodgridResultEnumerable identifiers = moodgridResult.Identifiers;
    foreach (GnMoodgridIdentifier identifier in identifiers)
    {
        string ident = identifier.MediaIdentifier;
        string group = identifier.Group;
        Console.WriteLine("\n\tX:" + x + " Y:" + y + " \nident:\t" + ident + " \ngroup:\t" + group
    );
    }
}
```

Java

```
/* Create a moodgrid presentation for the specified type */
GnMoodgridPresentation myPresentation = myMoodGrid.createPresentation(user, type);

/* Query the presentation type for its dimensions */
GnMoodgridDataPoint dataPoint = myMoodGrid.dimensions(type);
System.out.println("\n PRINTING MOODGRID " + dataPoint.getX() + " x " + dataPoint.getY() + " GRID
");

GnMoodgridPresentationDataIterator itr = myPresentation.moods().begin();

while(itr.hasNext()) {
    GnMoodgridDataPoint position = itr.next();

    /* Find the recommendation for the mood */
    GnMoodgridResult moodgridResult = myPresentation.findRecommendations(myProvider, position);

    System.out.println("\n\n\tX:" + position.getX()
        + "   Y:" + position.getY()
        + " name: " + myPresentation.moodName(position)
        + " count: " + moodgridResult.count()
        + " ");

    GnMoodgridResultIterator resultItr = moodgridResult.identifiers().begin();

    while(resultItr.hasNext()) {
        GnMoodgridIdentifier resultIdentifier = resultItr.next();
```

```
System.out.println("\n\tX:" + position.getX() + " Y:" + position.getY()+" ");
System.out.println("ident:\t" + resultIdentifier.mediaIdentifier()+" ");
System.out.println("group:\t" + resultIdentifier.group());

    }
}
```

Filtering Mood Results

You can use genre, origin, and era to filter Mood results. If you apply a filter, the results that are returned are pre-filtered, reducing the amount of data transmitted. For example, the following call sets a filter to limit results to tracks that fall within the Rock genre.

C++

Under construction

C#

Under construction

Java

Under construction

Best Practices and Design Requirements

Image Best Practices

Gracenote images – in the form of cover art, artist images and more – are integral features in many online music services, as well as home, automotive and mobile entertainment devices. Gracenote maintains a comprehensive database of images in dimensions to accommodate all popular applications, including a growing catalog of high-resolution (HD) images.

Gracenote carefully curates images to ensure application and device developers are provided with consistently formatted, high quality images – helping streamline integration and optimize the end-user experience. This topic describes concepts and guidelines for Gracenote images including changes to and support for existing image specifications.

Image Resizing Guidelines

Gracenote images are designed to fit

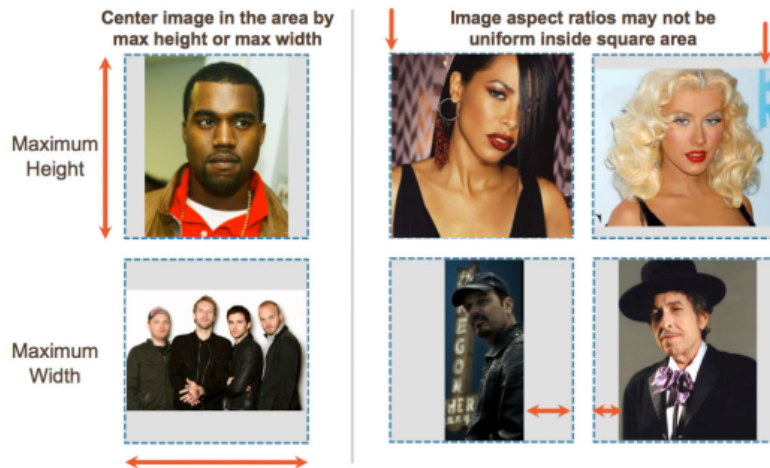
within squares as defined by the available image dimensions. This allows developers to present images in a fixed area within application or device user interfaces. Gracenote recommends applications center images horizontally and vertically within the predefined square dimensions, and that the square be transparent such that the background shows through. This results in a consistent presentation despite variation in the image dimensions. To ensure optimum image quality for end-users, Gracenote recommends that applications use Gracenote images in their provided pixel dimensions without stretching or resizing.

Gracenote resizes images based on the following guidelines:

- **Fit-to-square:** images will be proportionally resized to ensure their largest dimension (if not square) will fit within the limits of the next lowest available image size.

- **Proportional resizing:** images will always be proportionally resized, never stretched.
- **Always downscale:** smaller images will always be generated using larger images to ensure the highest possible image quality

Following these guidelines, all resized images will remain as rectangles retaining the same proportions as the original source images. Resized images will fit into squares defined by the available dimensions, but are not themselves necessarily square images.



i For Tribune Media Services (TMS) video images only, Gracenote will upsize images from their native size (288 x 432) to the closest legacy video size (300 x 450) – adhering to the fit-to-square rule for the 450 x 450 image size. Native TMS images are significantly closer to 300 x 450. In certain situations, downsizing TMS images to the next lowest legacy video size (160 x 240) can result in significant quality degradation when such downsized images are later displayed in applications or devices.

UI Best Practices for Audio Stream Recognition

The following are recommended best practices for applications that recognize streaming audio. Gracenote periodically conducts analysis on its MusicID-Stream product to evaluate its usage and determine if there are ways we can make it even better. Part of this analysis is determining why some MusicID-Stream recognition queries do not find a match.

Consistently Gracenote finds that the majority of failing queries contain an audio sample of silence, talking, humming, singing, whistling or live music. These queries fail because the Gracenote MusicID-Stream service can only match commercially released music.

Such queries are shown to usually originate from applications that do not provide good end user instructions on how to correctly use the MusicID-Stream service. Therefore Gracenote recommends application developers consider incorporating end user instructions into their applications from the beginning of the design phase. This section describes the Gracenote recommendations for instructing end users on how to use the MusicID-Stream service in order to maximize recognition rates and have a more satisfied user base.

This section is specifically targeted to applications running on a user's cellular handset, tablet computer, or similar portable device, although end user instructions should be considered for all applications using MusicID-Stream. Not all recommendations listed here are feasible for every application. Consider them options for improving your application and the experience of your end users.

Provide Clear and Accessible Instructions

Most failed recognitions are due to incorrect operation by the user. Provide clear and concise instructions to help the user correctly operate the application to result in a higher match rate and a better user experience. For example:

- Use pictures instead of text
- Provide a section in the device user manual (where applicable)
- Provide a help section within the application
- Include interactive instructions embedded within the flow of the application. For example, prompt the user to hold the device to the audio source.
- Use universal street sign images with written instructions to guide the user.

Provide a Demo Animation

Provide a small, simple animation that communicates how to use the application. Make this animation accessible at all times from the Help section.

Display a Progress Indicator During Recognition

When listening to audio, the application can receive status updates. The status updates indicate what percentage of the recording is completed. Use this information to display a progress bar (indicator) to notify the user.

Use Animations During Recognition

Display a simple image or animation that shows how to properly perform audio recognition, such as holding the device near the audio source if applicable.

Using Vibration, Tone, or Both to Indicate Recognition Complete

The user may not see visual notifications if they are holding the recording device up to an audio source. Also, the user may pull the device away from an audio source to check if recording has completed. This may result in a poor quality recording.

Display Help Messages for Failed Recognitions

When a recognition attempt fails, display a help message with a hint or tip on how to best use the MusicID-Stream service. A concise, useful tip can persuade a user to try again. Have a selection of help messages available; show one per failed recognition attempt, but rotate which message is displayed.

Allow the User to Provide Feedback

When a recognition attempt fails, allow the user to submit a hint with information about what they are looking for. Based on the response, the application could return a targeted help message about the correct

use of audio recognition.

Data Models

The following table links to GNSDK for Mobile data models for the corresponding query response type:

- **Music**
 - GnResponseAlbums
 - GnResponseTracks
 - GnResponseContributors

API Reference Documentation

You can develop GNSDK for Mobile applications using several object-oriented languages. The following table describes these APIs and where to find the corresponding API Reference documentation :

API Name	Description	Location in Package
Java (Android) API Reference	API descriptions of the GNSDK for Mobile Java Android interface	docs/html/Content/api_ref_java_android/html/index.html

Playlist PDL Specification

The GNSDK for Mobile Playlist Definition Language (PDL) is a query syntax, similar to Structured Query Language (SQL), that enables flexible custom playlist generation using human-readable text strings. PDL allows developers to dynamically create custom playlists. By storing individual PDL statements, applications can create and manage multiple preset and user playlists for later use.

PDL statements express the playlist definitions an application uses to determine what items are included in resulting playlists. PDL supports logic operators, operator precedence and in-line arithmetic. PDL is based on Search Query Language (SQL). This section assumes you understand SQL and can write SQL statements.



NOTE: Before implementing PDL statement functionality for your application, carefully consider if the provided More Like This function, `gnsdk_playlist_generate_morelikethis()` meets your design requirements. Using the More Like This function eliminates the need to create and validate PDL statements.

PDL Syntax

This topic discusses PDL keywords, operators, literals, attributes, and functions.

Note: Not all keywords support all operators. Use `gnsdk_playlist_statement_validate()` to check a PDL Statement, which generates an error for invalid operator usage.

Keywords

PDL supports these keywords:

Keyword	Description	Required or Optional	PDL Statement Order
GENERATE PLAYLIST	All PDL statements must begin with either GENERATE PLAYLIST or its abbreviation, GENPL	Required	1
WHERE	Specifies the attributes and threshold criteria used to generate the playlist. If a PDL statement does not include the WHERE keyword, Playlist operates on the entire collection.	Optional	2

Keyword	Description	Required or Optional	PDL Statement Order
ORDER	<p>Specifies the criteria used to order the results' display.</p> <p>If a PDL statement does not include the ORDER keyword, Playlist returns results in random order.</p> <p>Example: Display results in based on a calculated similarity value; tracks having greater similarity values to input criteria display higher in the results.</p> <p>The expression format is: <identifier> <operator> <identifier></p>	Optional	3
LIMIT	<p>Specifies criteria used to restrict the number of returned results.</p> <p>Also uses the keywords RESULT and PER.</p> <p>Example: Limiting the number of tracks displayed in a playlist to 30 results with a maximum of two tracks per artist.</p> <p>The expression format is: <identifier> <operator> <identifier></p>	Optional	4
SEED	<p>Specifies input data criteria from one or more ids. Typically, a Seed is the This in a More Like This playlist request.</p> <p>Example: Using a Seed of Norah Jones' track Don't Know Why to generate a playlist of female artists of a similar genre.</p> <p>The expression format is: <identifier> <operator> <identifier></p>	Optional	NA

Example: Keyword Syntax

This PDL statement example shows the syntax for each keyword. In addition to <att_imp>, <expr>, and <score> discussed above, this example shows:

- <math_op> is one of the valid PDL mathematical operators.
- <number> is positive value.
- <attr_name> is a valid attribute, either a Gracenote-delivered attribute or an implementation-specific attribute.

```

GENERATE PLAYLIST
WHERE <att_imp> [<math_op> <score>][ AND|OR <att_imp>]
ORDER <expr>[ <math_op> <expr>]
LIMIT [number RESULT | PER <attr_name>][,number [ RESULT | PER <attr_name>]]

```

Operators

PDL supports these operators:

Operator Type	Available Operators
Comparison	>, >=, <, <=, ==, !=, LIKE LIKE is for fuzzy matching, best used with strings; see PDL Examples
Logical	AND, OR
Mathematical	+, -, *, /

Literals

PDL supports the use of single (') and double (") quotes, as shown:

- Single quotes: 'value one'
- Double quotes: "value two"
- Single quotes surrounded by double quotes: "'value three'"



You must enclose a literal in quotes or Playlist evaluates it as an attribute.

Attributes

Most attributes require a Gracenote-defined numeric identifier value or GDO (GnDataObject in object-oriented languages) for Seed.

- Identifier: Gracenote-defined numeric identifier value is typically a 5-digit value; for example, the genre identifier 24045 for Rock. These identifiers are maintained in lists in Gracenote Service; download the lists using GNSDK for Mobile Manager's Lists and List Types APIs
- GDO Seed: Use GNSDK Manager's GDO APIs to access XML strings for Seed input.

The following table shows the supported system attributes and their respective required input. The first four attributes are the GOET attributes.



The delivered attributes have the prefix GN_ to denote they are Gracenote standard attributes. You can extend the attribute functionality in your application by implementing custom attributes; however, do not use the prefix GN_.

Name	Attribute	Required Input
Genre Origin Era Artist Type Mood Tempo	GN_Genre GN_Origin GN_Era GN_ArtistType GN_Mood GN_Tempo	Gracenote-defined numeric identifier value GDO Seed XML string
Artist Name Album Name	GN_ArtistName GN_AlbumName	Text string

Functions

PDL supports these functions:

- RAND(max value)
- RAND(max value, seed)

PDL Statements

This topic discusses PDL statements and their components.

Attribute Implementation <att_imp>

A PDL statement is comprised of one or more attribute implementations that contain attributes, operators, and literals. The general statement format is:

"GENERATE PLAYLIST WHERE <attribute> <operator> <criteria>"

You can write attribute implementations in any order, as shown:

GN_ArtistName == "ACDC"

or

"ACDC" == GN_ArtistName

WHERE and ORDER statements can evaluate to a score; for example:

"GENERATE PLAYLIST WHERE LIKE SEED > 500"

WHERE statements that evaluate to a non-zero score determine what ids are in the playlist results.

ORDER statements that evaluate to a non-zero score determine how ids display in the playlist results.

Expression <expr>

An expression performs a mathematical operation on the score evaluated from a attribute implementation.

[<number> <math_op>] <att_imp>

For example:

3 * (GN_Era LIKE SEED)

Score <score>

Scores can range between -1000 and 1000.

For boolean evaluation, True equals 1000 and False equals 0.

Note: For more complex statement scoring, concatenate attribute implementations and add weights to a PDL statement.

Example: PDL Statements

The following PDL example generates results that have a genre similar to and on the same level as the seed input. For example, if the Seed genre is a Level 2: Classic R&B/Soul, the matching results will include similar Level 2 genres, such as Neo-Soul.

```
"GENERATE PLAYLIST WHERE GN_Genre LIKE SEED"
```

This PDL example generates results that span a 20-year period. Matching results will have an era value from the years 1980 to 2000.

```
"GENERATE PLAYLIST WHERE GN_Era >= 1980 AND GN_Era < 2000"
```

This PDL example performs fuzzy matching with Playlist, by using the term LIKE and enclosing a string value in single (') or double (") quotes (or both, if needed). It generates results where the artist name may be a variation of the term ACDC, such as:

- ACDC
- AC/DC
- AC*DC

```
"GENERATE PLAYLIST WHERE (GN_ArtistName LIKE 'ACDC')"
```

The following PDL example generates results where:

- The tempo value must be less than 120 BPM.
- The ordering displays in descending order value, from greatest to least (119, 118, 117, and so on).
- The genre is similar to the Seed input.

```
"GENERATE PLAYLIST WHERE GN_Tempo > 120 ORDER GN_Genre LIKE SEED"
```

GNSDK Glossary

A

Album Name (in TLS)

Used by Gracenote Three-Line-Solution (TLS) for classical music. In most cases, a classical album's title is comprised of the composer(s) and work(s) that are featured on the product, which yields a single entity in the album name display. However, for albums that have formal titles (unlike composers and works), the title is listed as it is on the product. General title example: Beethoven: Violin Concerto. Formal title example: The Best Of Baroque Music.

Android Activity Object

An Activity Object is one of the building blocks of an Android application. An Activity provides the application with context to render to the screen. An Android application typically has at least one Activity object.

Artist Name (in TLS)

Used by Gracenote Three-Line-Solution (TLS) for classical music. A consistent format is used for listing a recording artist(s)—by soloist(s), conductor, and ensemble(s)—which yields a single entity in the artist name display. For example: Hilary Hahn; David Zinman: Baltimore Symphony Orchestra

C

Client ID

Each customer receives a unique Client ID string from Gracenote. This string uniquely identifies each application and lets Gracenote deliver the specific features for which the application is licensed. The Client ID string has the following format: 123456-789123456789012312. The first part is a six-digit Client ID, and the second part is a 17-digit Client ID Tag.

Composer (in TLS)

Used by Gracenote Three-Line-Solution (TLS) for classical music. The composer(s) that are featured on an album are listed by their last name in the album title (where applicable). In the examples noted below, the composer is Beethoven.

Contributor

A Contributor refers to any person who plays a role in an AV Work. Actors, Directors, Producers, Narrators, and Crew are all consider a Contributor.

Popular recurring Characters such as Batman, Harry Potter, or Spider-man are also considered Contributors in Video Explore.

Credit

A credit lists the contribution of a person (or occasionally a company, such as a record label) to a recording. Generally, a credit consists of: The name of the person or company. The role the person played on the recording (an instrument played, or another role such as composer or producer). The tracks affected by the contribution. A set of optional notes (such as “plays courtesy of Capitol records”). See Role

F

Features

Term used to encompass a particular media stream's characteristics and attributes; this is metadata and information accessed from processing a media stream. For example, when submitting an Album's Track, this includes information such as fingerprint, mood, and tempo metadata (Attributes).

Fingerprint

A unique representation of a sample of audio that can be used to identify a track.

G

Generation Criterion

A selection rule for determining which tracks to add to a playlist.

Generator

Short for playlist generator. This is a term used to indicate that an artist has at least one of the extended metadata descriptors populated. The metadata may or may not be complete, and the artist text may or may not be locked or certified. See also: extended metadata, playlist optimized artist.

Genre

A categorization of a musical composition characterized by a particular style.

L

Link Module

A module available in the Desktop and Auto products that allows applications to access and present enriched content related to media that has been

identified using identification features.

M

Manual Playlist

A manual playlist is a playlist that the user has created by manually selecting tracks and a play order. An auto-playlist is a playlist generated automatically by software. This distinction only describes the initial creation of the playlist; once created and saved, all that matters to the end-user is whether the playlist is static (and usually editable) or dynamic (and non-editable). All manual playlists are static; the track contents do not change unless the end-user edits the playlist. An auto-playlist may be static or dynamic.

Metadata

Data about data. For example, metadata such as the artist, title, and other information about a piece of digital audio such as a song recording.

Mood

Track-level perceptual descriptor of a piece of music, using emotional terminology that a typical listener might use to describe the audio track; includes hierarchical categories of increasing granularity. See Sonic Attributes.

More Like This

A mechanism for quickly generating playlists from a user's music collection based on their similarity to a designated seed track, album, or artist.

Multiple Match

During CD recognition, the case where a single TOC from a CD may have more than one exact match in the MDB. This is quite rare, but can happen. For example with CDs that have only one track, it is possible that two of these one-track CDs may have exactly the same length (the exact same number of frames). There is no way to resolve these cases automatically as there is no other information on the CD to distinguish between them. So the user must be presented with a dialog box to allow a choice between the alternatives. See also: frame, GN media database, TOC.

MusicID Module

Enables MusicID recognition for identifying CDs, digital music files and streaming audio and delivers relevant metadata such as track titles, artist names, album names, and genres. Also provides library organization and direct lookup features.

MusicID-File Module

A feature of the MusicID product that identifies digital music files using a combination of waveform analysis technology, text hints and/or text lookups.

P

PCM (Pulse Coded Modulation)

Pulse Coded Modulation is the process of converting an analog in a sequence of digital numbers. The accuracy of conversion is directly affected by the sampling frequency and the bit-depth. The sampling frequency defines how often the current level of the audio signal is read and converted to a digital number. The bit-depth defines the size of the digital number. The higher the frequency and bit-depth, the more accurate the conversion.

PCM Audio

PCM data generated from an audio signal.

Playlist

A set of tracks from a user's music collection, generated according to the criteria and limits defined by a playlist generator.

Popularity

Popularity is a relative value indicating how often metadata for a track, album, artist and so on is accessed when compared to others of the same type. Gracenote's statistical information about the popularity of an album or track, based on aggregate (non-user-specific) lookup history maintained by Gracenote servers. Note that there's a slight difference between track popularity and album popularity statistics. Track popularity information identifies the most popular tracks on an album, based on text lookups. Album popularity identifies the most frequently looked up albums, either locally by the end-user, or globally across all Gracenote users. See Rating and Ranking

R

Ranking

For Playlist, ranking refers to any criteria used to order songs within a playlist. So a playlist definition (playlist generator) may rank songs in terms of rating values, or may rank in terms of some other field, such as last-played date, bit rate, etc.

Rating

Rating is a value assigned by a user for the songs in his or her collection. See Popularity and Ranking.

Role

A role is the musical instrument a contributor plays on a recording. Roles can also be more general, such as composer, producer, or engineer. Gracenote has a specific list of supported roles, and these are broken into role categories, such as string instruments, brass instruments. See Credit.

S ---

Seed Track, Disc, Artist, Genre

Used by playlist definitions to generate a new playlist of songs that are related in some way, or similar, to a certain artist, album, or track. This is a term used to indicate that an artist has at least one of the extended metadata descriptors populated. The metadata may or may not be complete, and the artist text may or may not be locked or certified. See also: extended metadata, playlist optimized artist, playlist-related terms.

Sonic Attributes

The Gracenote Service provides two metadata fields that describe the sonic attributes of an audio track. These fields, mood and tempo, are track-level descriptors that capture the unique characteristics of a specific recording. Mood is a perceptual descriptor of a piece of music, using emotional terminology that a typical listener might use to describe the audio track. Tempo is a description of the overall perceived speed or pace of the music. The Gracenote mood and tempo descriptor systems include hierarchical categories of increasing granularity, from very broad parent categories to more specific child categories.

T ---

Target Field

The track attribute used by a generation criterion for selecting tracks to include in a generated playlist.

Tempo

Track-level descriptor of the overall perceived speed or pace of the music; includes hierarchical categories of increasing granularity. See Sonic Attributes.

Three-Line-Solution (TLS)

GNSDK support for classical music metadata. Gracenote TLS provides the four basic classical music metadata components—composer, album name, artist name, and track name.

TLS

See Three-Line-Solution.

TOC

Table of Contents. An area on CDs, DVDs, and Blu-ray discs that describes the unique track layout of the disc.

Track Name (in TLS)

Used by Gracenote Three-Line-Solution (TLS) for classical music. A consistent format is used for listing a track title—composer, work title, and (where applicable) movement title—which yields a single entity in the track name display. For example: Beethoven: Violin Concerto In D, Op. 61 – 1. Allegro Ma Non Troppo

Index

A

ACR 5, 49

Album 6, 10, 13, 15, 19, 23, 41, 75, 105, 116, 131

AlbumID 16, 18, 77

Android 37-39, 41-43, 45, 47, 51, 57, 64, 70, 78, 91, 99, 127

API Reference 127

Artist 6, 10, 19-20, 41, 75, 116, 131

 Image 75

 Name 10, 116, 131

Audio 6, 13

B

Behavior 54

Best Practices 17, 55, 75, 122-123

Bundles 63

C

Callback 57

CD TOC 2, 18, 60, 69

Classical Music 9, 93

Client ID 43, 45-46

Client Tag 48

Collection 21-22, 24, 37, 104

Contributor 6

Cover Art 20, 36

D

Data Model 39, 93, 97

Database 39

Descriptor 50

Dimensions 20

Discovery 21, 104

DSP 3, 15, 35

E

Enriched Content 19, 98

Enriched Metadata 8

Enrichment 36

Entries 40

Environment 58

EPG 5, 52

Equivalency 22

Era 75, 109, 117, 132

F

FileInfo 73

Find 120

Fingerprint 15, 41, 67

G

GDB 52

GDO 18, 23, 105, 116, 130

Genre 6, 13, 20, 22, 30, 75, 116, 131

Getting Started 38, 41

GnMic 69, 90

GNSDK for Mobile i, 6, 13-15, 21-23, 35, 41-42, 113, 126-128

GOET 116, 130

Gracenote i, 1, 6, 8-9, 11-12, 14-15, 18-19, 21, 23-26, 36-38, 41-42, 45-46, 50, 57, 59, 63, 67, 69, 72, 80, 90, 96, 98, 108, 115, 118, 122-123, 129

Gracenote Media Elements 6

Groups 53

H

Header Files 46

Hierarchical Groups 8

I

Identifiers 23

Identify 4, 18

Image

Dimensions 20

Formats 19

Initialize 45, 48, 65, 73, 85, 91, 106

Instantiate 45-46, 57, 70, 77, 91, 106

Introduction 1, 41

iOS 90

Iterate 111

L

LANG 93

Language 4, 21, 50, 107, 113, 128

LibraryID 16, 77

License File 47

Link 2, 11, 19, 102

Linux 36

List 10, 79

Literals 115, 130

Local 2, 59, 63, 67, 75, 99

Local Lookup 61

Local Storage 106

Locale 50, 96

Locale-Dependent 51, 97

Log 64, 84

M

Manager 2, 116, 119, 130
Match 3, 14
Memory 35, 62
Metadata 9, 16, 23, 78, 93, 95
Module 12
Mood 2, 9, 23-26, 30, 36, 75, 118
MoodGrid 2
Multi-Threaded Access 53
Multiple TOC Matches 14
Multiple TOCs and Fuzzy Matching 14
Music 1-2, 6, 8, 10, 12, 19-20, 36, 42, 50, 69, 77, 90, 97, 126
MusicID 3, 12, 14-15, 17, 22-23, 25, 36, 46, 69, 75, 91, 105, 119
MusicID-File 3, 15, 17, 69, 77, 106
MusicID-Stream 2, 18, 63, 90, 123

N

Navigating 26
Needs Decision 96
Number 22

O

Order 114, 128
Origin 75, 110, 116, 131
Overview 1, 77, 118

P

Parse 46
Partial 96
PCM 40, 72, 91
PDL 24, 108, 113, 128
Performance 112

Permissions 38

Physical Media 113

Playlist 4, 9, 21, 23, 29, 36, 53, 104, 113, 119, 128

Playlist Description Language (PDL) 104

Populate 104

Process 70

Product 14, 84

Project 38, 42

Providers 118

Q

QDB 66

Query 60, 68, 93, 98, 108, 120

R

Radio 3

RAM 35, 54

Region 50

Requirements 21, 36

Results 29, 78, 91, 109, 122

Rhythm 4, 33

S

Sample Application 38-39, 41-43

Save 103

Score 117, 132

Search 75

Seed 24, 115, 129

Setting Up 37, 41-42

Size 40, 99

SQLite 2, 22, 39, 53, 59, 63, 70, 106, 119

Station 4

Status 50, 78

Storage 22, 24, 35, 49, 58-59

Storage Provider 54, 66

Streaming Audio 93

Submit 2

System Requirements 35

T

Taste 4

Tempo 9, 75, 118, 132

Text 3, 13, 15, 36, 69, 71, 116, 131

Third-Party

Identifiers 11

Title 9, 75, 100

Track 6, 9, 15, 23, 41, 71, 75, 105

TrackID 17, 77

U

Update 53

User 45, 71, 77, 91, 96, 124

History 39

Object 45-46

V

Values 25

Video 5

VideoID 5

Voice Commands 29

W

Windows 37, 42, 57, 61, 66

Windows CE 61

Windows Phone 47, 51, 60, 66, 71, 79, 92, 100

Work 9