



GNSDK OO APIs

Developer Guide

Release: 3.08.6.5437

Published: 9/14/2017 1:08 PM

Copyright 2017 Gracernote, Inc.. All rights reserved.

Information in this document is subject to change without notice. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of those agreements. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or any means electronic or mechanical, including photocopying and recording for any purpose other than the purchaser's personal use without the written permission of Gracernote, Inc..

Gracernote, Inc.

Contents

Chapter 1 Concepts	14
1.1 About Gracenote	14
1.2 What is the GNSDK?	14
1.3 Gracenote Media Elements	15
1.3.1 Genre and Other List-Dependent Values	15
1.3.1.1 Genres and List Hierarchies	16
1.3.1.2 Simplified and Detailed Hierarchical Groups	18
1.3.2 Core and Enriched Metadata	18
1.3.3 Mood and Tempo (Sonic Attributes)	19
1.3.4 Classical Music Metadata	19
1.3.5 Third-Party Identifiers and Preferred Partners	22
1.4 Music Modules	22
1.4.1 MusicID Overview	23
1.4.1.1 CD TOC Recognition	24
TOC Identification	24
Multiple TOC Matches	25
Multiple TOCs and Fuzzy Matching	25
1.4.1.2 Text-Based Recognition	26
1.4.1.3 Fingerprint-Based Recognition	26
About DSP	27
1.4.2 MusicID-File Overview	27
1.4.2.1 Waveform and Metadata Recognition	27
1.4.2.2 Advanced Processing Methods	28
LibraryID	28

AlbumID	29
TrackID	29
MusicID-File Best Practices	29
Usage Notes	30
1.4.2.3 MusicID vs. MusicID-File	30
1.4.3 MusicID Stream	31
1.4.4 Music Enrichment	32
1.4.5 Playlists	32
1.4.5.1 Collection Summaries	33
1.4.5.2 More Like This	33
1.4.5.3 .Playlist Requirements and Recommendations	33
Simplified Playlist Implementation	33
Playlist Content Requirements	34
Playlist Storage Recommendations	34
Playlist Resource Requirements	34
Playlist Level Equivalency for Hierarchical Attributes	35
1.4.5.4 Key Playlist Components	35
Media metadata: Metadata of the media items (or files)	36
Attributes: Characteristics of a media item, such as Mood or Tempo	36
Unique Identifiers	36
Collection Summary	37
Storage	37
Seed	37
Playlist generation	37
1.4.5.5 Mood Overview	38
Mood Descriptors	38
Mood Valence/Arousal Model	38
Mood Levels	39
1.4.5.6 Level 1 Valence/Arousal Map	40
1.4.5.7 Level 2 Valence/Arousal Map	40
Navigating with Mood	40
Slider Navigation	41

Grid Navigation	41
Bubble Magnitude Navigation	42
Other Mood Design Considerations	43
Mood Level Arousal/Valance Values	44
Level 1 Mood Levels	44
Level 2 Mood Levels	46
1.4.6 Rhythm Overview	50
1.4.7 MusicID-Match Overview	52
1.5 Image Formats and Dimensions	52
1.5.1 Available Image Dimensions	53
1.5.2 Common Media Image Dimensions	53
1.5.2.1 Music Cover Art	53
1.5.2.2 Artist Images	54
1.5.2.3 Genre Images	54
1.5.2.4 Video Cover Art	54
Video Image Dimension Variations	54
Video (AV Work) Images	55
Video Product Images	55
Video Contributor Images	55
1.5.3 Image Best Practices	55
1.5.3.1 Using a Default Image When Cover Art Is Missing	56
1.5.3.2 Image Resizing Guidelines	56
Chapter 2 Setup and Samples	58
2.1 Modules Overview	58
2.1.1 Modules in the GNSDK Package	60
2.1.1.1 GNSDK Modules	60
2.2 Basic Application Steps	61
2.3 Setup and Initialization	62
2.3.1 Authorizing a GNSDK Application	62
2.3.1.1 Client ID/Tag	63
2.3.1.2 License File	63
2.3.2 Including Header Files	64
2.3.3 Instantiating a GNSDK Manager Object	64

2.3.3.1 Specifying the License File	65
2.3.4 Instantiating a User Object	66
2.3.4.1 Saving the User Object to Persistent Storage	67
2.3.4.2 User Object Options	69
Chapter 3 Develop and Implement	72
3.1 About this Documentation	72
3.2 Loading a Locale	72
3.2.1 Locale-Dependent Data	75
3.2.2 Default Regions and Descriptors	76
3.2.3 Locale Groups	77
3.2.3.1 Multi-Threaded Access	78
3.2.3.2 Updating Locales and Lists	78
Update Notification	78
3.2.4 Locale Behavior	79
3.2.5 Best Practices	80
3.3 Using Logging	83
3.3.1 Enabling GNSDK Logging	84
3.3.2 Implementing Callback Logging	86
3.4 Enabling and Using GNSDK Storage	86
3.4.1 Enabling a Provider for GNSDK Storage	87
3.4.2 GNSDK Production Stores	88
3.4.3 GNSDK Databases	89
3.4.4 Setting GNSDK Storage Folder Locations	89
3.4.5 Storing Results in Local Databases	90
3.4.6 Getting Results Database Information	91
3.4.6.1 Image Information	91
3.4.6.2 Database Versions	92
3.4.6.3 Getting Available Locales	92
3.4.7 Setting Online Cache Expiration	92
3.4.8 Managing Online Cache Size and Memory	92
3.4.9 Closing and Deleting a Results Database	93
3.4.10 Managing Results Database Size	93

3.5 Setting Local and Online Lookup Modes	94
3.5.1 Supported Lookup Modes	95
3.5.2 Default Lookup Mode	96
3.5.3 Setting the Lookup Mode for a User or Queries	96
3.5.4 Using Both Local and Online Lookup Modes	97
3.6 Using a Local Fingerprint Database	98
3.6.1 Downloading and Ingesting Bundles	99
3.6.2 Designating a Storage Provider	103
3.7 Identifying Music	104
3.7.1 Identifying Music Using a CD-TOC, Text, or Fingerprints (MusicID)	105
MusicID Queries	105
Options for MusicID Queries	106
Identifying Music Using a CD TOC	107
Identifying Music Using Text	108
Identifying Music Using Fingerprints	110
PCM Audio Format Recommendations	111
MusicID Fingerprinting	111
3.7.1.1 Best Practices for MusicID Text Match Queries	120
3.7.1.2 Identifying Music in a Batch Query	121
Batch Query Code Sample (C++)	122
3.7.2 Identifying Audio Files (MusicID-File)	123
3.7.2.1 Implementing an Events Delegate	125
3.7.2.2 Adding Audio Files for Identification	133
Setting Audio File Identification	133
MusicID-File Fingerprinting	137
3.7.2.3 Setting Options for MusicID-File Queries	144
3.7.2.4 Making a MusicID-File Query	145
Options When Making Query Call	145
3.7.3 Identifying Streaming Music (MusicID Stream)	146
3.7.3.1 Implementation Notes	148
3.7.3.2 Setting Options for Streaming Audio Queries	149
3.7.3.3 Music-ID Stream Code Samples	150

3.8 Processing Returned Metadata Results	158
3.8.1 Needs Decision	158
3.8.2 Full and Partial Metadata Results	159
3.8.3 Locale-Dependent Data	161
3.8.4 Accessing Enriched Content using Asset Fetch	161
3.8.4.1 Setting the Query Option for Enriched Content	161
3.8.4.2 Processing Enriched Content	162
3.8.4.3 Retrieving a Content Asset	163
Retrieving Local or Online Content	163
3.8.4.4 Retrieving and Parsing Enriched Content Code Samples	164
3.8.5 Accessing Classical Music Metadata	170
3.8.5.1 Example Classical Music Output	172
3.9 Generating a Playlist	175
3.9.1 Creating a Collection Summary	176
3.9.2 Populating a Collection Summary	177
3.9.2.1 Retrieving Playlist Attributes in Queries	178
3.9.3 How Playlist Gathers Data	178
3.9.4 Working with Local Storage	179
3.9.5 Generating a Playlist Using More Like This	180
3.9.6 Generating a Playlist Using PDL (Playlist Description Language)	183
3.9.7 Accessing Playlist Results	184
3.9.8 Working with Multiple Collection Summaries	188
3.9.8.1 Join Performance and Best Practices	189
3.9.9 Synchronizing Collection Summaries	189
3.9.9.1 Iterating the Physical Media	190
3.9.9.2 Processing the Collection	190
3.9.10 Implementing Mood	191
3.9.10.1 Prerequisites	192
3.9.10.2 Enumerating Data Sources using Mood Providers	192
3.9.10.3 Creating and Populating a Mood Presentation	193
3.9.10.4 Iterating Through a Mood Presentation	194

3.9.10.5 Filtering Mood Results	197
3.9.11 Playlist PDL Specification	197
3.9.11.1 PDL Syntax	197
Keywords	198
Example: Keyword Syntax	199
Operators	200
Literals	200
Attributes	200
Functions	201
PDL Statements	201
Attribute Implementation <att_imp>	201
Expression <expr>	202
Score <score>	202
3.9.11.2 Example: PDL Statements	202
3.10 Implementing Rhythm	203
3.10.1 Creating a Rhythm Query	204
3.10.2 Setting Rhythm Query Options	205
3.10.3 Generating Recommendations	206
3.10.4 Creating a Radio Station and Playlist	206
3.10.5 Providing Feedback	207
3.10.6 Tuning a Radio Station	208
3.10.7 Saving and Retrieving Radio Stations	209
3.11 Best Practices and Design Requirements	209
3.11.1 Managing Image Dimensions	209
3.11.1.1 Image Resizing Guidelines	210
3.11.1.2 Image Formats and Dimensions	211
3.11.1.3 Common Media Image Dimensions	212
Music Cover Art	212
Video Cover Art	212
Artist Imagery	213
3.11.1.4 Variations in Video Image Dimensions	213
AV Work Images	213
Video Product Images	213

Contributor Images	214
3.11.1.5 TV Channel Logo Sizes	214
3.11.1.6 TV Program and Category Image Sizes	214
3.11.2 Collaborative Artists Best Practices	215
3.11.2.1 Handling Collaborations when Processing a Collection	215
3.11.2.2 Displaying Collaborations during Playback	216
3.11.2.3 Displaying Collaborations in Navigation	216
3.11.2.4 Handling Collaborations in Playlists	217
3.11.3 UI Best Practices for Audio Stream Recognition	217
3.11.3.1 Provide Clear and Accessible Instructions	218
3.11.3.2 Provide a Demo Animation	219
3.11.3.3 Display a Progress Indicator During Recognition	219
3.11.3.4 Use Animations During Recognition	219
3.11.3.5 Using Vibration, Tone, or Both to Indicate Recognition Complete	219
3.11.3.6 Display Help Messages for Failed Recognitions	219
3.11.3.7 Allow the User to Provide Feedback	219
3.12 Deploying Android Applications	220
3.12.1 GNSDK Android Permissions	220
Chapter 4 Data Models	221
4.1 Data Models	221
Chapter 5 API Reference	223
5.1 API Reference Overview	224
5.2 Playlist PDL Specification	224
5.2.1 PDL Syntax	225
5.2.1.1 Keywords	225
Example: Keyword Syntax	227
5.2.1.2 Operators	227
5.2.1.3 Literals	227
5.2.1.4 Attributes	228
5.2.1.5 Functions	228

5.2.1.6 PDL Statements	229
Attribute Implementation <att_imp>	229
Expression <expr>	229
Score <score>	230
5.2.2 Example: PDL Statements	230
Glossary	232
Index	242

This page intentionally left blank to

Chapter 1 Concepts

This section introduces GNSDK, and presents important product concepts.

1.1 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.

1.2 What is the GNSDK?

Gracenote SDK (GNSDK) is a platform that delivers Gracenote technologies to devices, desktop applications, web sites, and back-end 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 multi-threaded environments. GNSDK is also lightweight—made to run in desktop applications and even to support popular portable devices.

1.3 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:

Music

- Music CD
- Album
- Track
- Artist
- Contributor

Video

- Video Product (DVD/Blu-Ray)
- AV Work
- Contributor
- Series
- Season

1.3.1 Genre and Other List-Dependent Values

GNSDK 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, and roles.

List-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. These kinds of list-based metadata values are called *locale-dependent*.

1.3.1.1 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.

1.3.1.2 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.

Below are examples of a Simplified and Detailed Genre Hierarchy Groups. The values below are for documentation purposes only. Please contact your Gracenote representative for more information.

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

Contact your Gracenote representative for more detail.

1.3.2 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.

For music, core metadata for albums, tracks, and artists includes:

- Genres
- Origins
- Era
- Title
- Types
- External IDs

Enriched metadata includes album cover art, artist images, and more.

1.3.3 *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.

To use this feature, your application requires special metadata entitlements. Contact your Gracenote representative for more information.

1.3.4 *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>
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	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. Example: Hilary Hahn; David Zinman: Baltimore Symphony Orchestra
Catalog #	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)."
Composer	The composer(s) that are featured on an album are listed by their last name in the album title (where applicable)
Key	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. 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.)
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>
------------	--

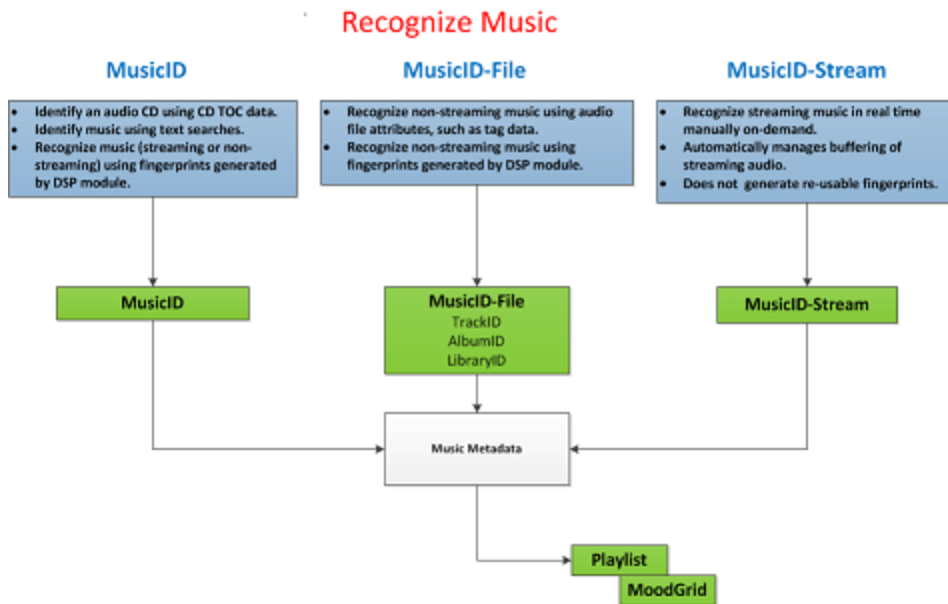
1.3.5 Third-Party Identifiers and Preferred Partners

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.

1.4 Music Modules

The following diagram shows the kinds of identification queries each music module supports.



Access Metadata

1.4.1 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
- Mood and tempo descriptors

- Music Enrichment content, including cover art, artist images, biographies, and reviews

GNSDK 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)

1.4.1.1 CD TOC Recognition

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.

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.

1.4.1.2 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:

- 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.

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. Also see [Identifying Music Using Text](#) and [Best Practices for MusicID Text Match Queries](#).

1.4.1.3 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 modules. This module is optional unless the application performs music identification or generates audio features for submission to Gracenote.

1.4.2 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.

1.4.2.1 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.

1.4.2.2 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.



Note: 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.

1.4.2.3 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).



Note: 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.

1.4.3 *MusicID Stream*

You can use MusicID Stream to recognize music delivered as a continuous stream. Specifically, MusicID Stream enables these features:

- Recognizes streaming music in real time, on-demand (user-initiated).
- 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. 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, the system waits until enough audio is received. The identification process spawns a thread and completes asynchronously.

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

1.4.4 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.

Examples of enriched content are:

- Album cover art
- Artist images
- Album reviews
- Artist biographies

1.4.5 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 to generate highly relevant playlists.

1.4.5.1 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.

1.4.5.2 More Like This

More Like This is a powerful and popular navigation feature made possible by Gracenote and GNSDK 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.

1.4.5.3 .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'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 36*.

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 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	GNSDK 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 19. 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.

1.4.5.4 Key Playlist Components

Playlist operates on several key components. The GNSDK 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 operation, and consequently must be implemented with one or both of GNSDK'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.



Note: 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 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 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 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.

1.4.5.5 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.

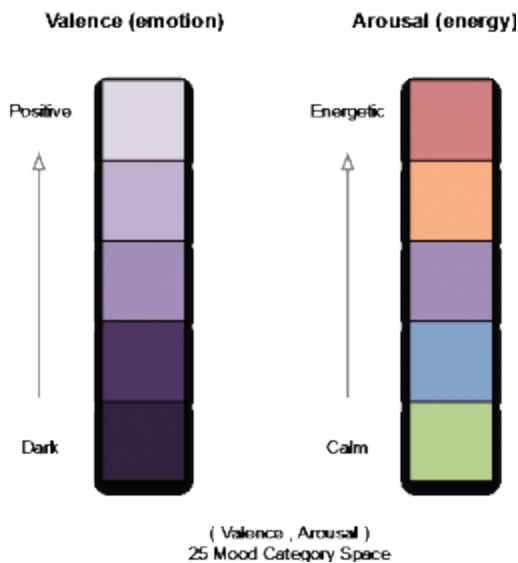
Mood Descriptors

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](#).

1.4.5.6 Level 1 Valence/Arousal Map

L1 (35) Category Sonic Mood Grid

		Arousal →				
		1	2	3	4	5
Valence ↓	1	Somber	Gloomy	Serious	Brooding	Aggressive
	2	Melancholy	Eerie	Yearning	Urgent	Defiant
	3	Sentimental	Supersaturated	Sensual	Fairy	Emerging
	4	Tender	Romantic	Empowering	Biting	Rowdy
	5	Peaceful	Easygoing	Upbeat	Lively	Excited

1.4.5.7 Level 2 Valence/Arousal Map

L2 (100) Category Sonic Mood Grid

		Arousal →				
		1	2	3	4	5
Valence ↓	1	Dark Cosmic	Creepy / Ominous	Depressed / Lonely	Gloomy / Soulful	Serious / Cerebral
	2	Solemn / Spiritual	Enigmatic / Mysterious	Sober / Determined	Strumming Yearning	Melodramatic
	3	Wistful / Forlorn	Sad / Soulful	Cool Confidence	Dark Groovy	Sensitive / Exploring
	4	Mysterious / Dreamy	Light Melancholy	Casual Groove	Wary / Defiant	Bittersweet Pop
	5	Lyrical Sentimental	Cool Melancholy	Intimate Bittersweet	Smoky / Romantic	Dreamy Pulse
	6	Tender / Sincere	Gentle Bittersweet	Suave / Sultry	Dark Playful	Soft Soulful
	7	Romantic / Lyrical	Light Groovy	Dramatic / Romantic	Lush / Romantic	Dramatic Emotion
	8	Refined / Mannered	Awakening / Statly	Sweet / Sincere	Heartfelt Passion	Strong / Stable
	9	Reverent / Healing	Quiet / Introspective	Friendly	Charming / Easygoing	Soulful / Easygoing
	10	Pastoral / Serene	Delicate / Tranquil	Hopeful / Breezy	Cheerful / Playful	Carefree Pop
Valence ↓	6	Thrilling	Dreamy Brooding	Alienated / Brooding	Chaotic / Intense	Aggressive Power
	7	Hypnotic Rhythm	Evocative / Intriguing	Energetic Melancholy	Dark Hard Beat	Heavy Triumphant
	8	Energetic Dreamy	Dark Urgent	Energetic Anxious	Attitude / Defiant	Hard Dark Excitement
	9	Energetic Yearning	Dark Pop	Dark Pop Intensity	Heavy Brooding	Hard Positive Excitement
	10	Intimate	Passionate Rhythm	Energetic Abstract Groove	Edgy / Sexy	Abstract Beat
	1	Sensual Groove	Dark Sparkling Lyrical	Fiery Groove	Arousing Groove	Heavy Beat
	2	Idealistic / Stirring	Focused Sparkling	Triumphant / Rousing	Confident / Tough	Driving Dark Groove
	3	Powerful / Heroic	Invigorating / Joyous	Jubilant / Soulful	Ramshackle / Rollicking	Wild / Rowdy
	4	Happy / Soulful	Playful / Swingin'	Exuberant / Festive	Upbeat Pop Groove	Happy Excitement
	5	Party / Fun	Showy / Rousing	Lusty / Jaunty	Loud Celebratory	Euphoric Energy

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

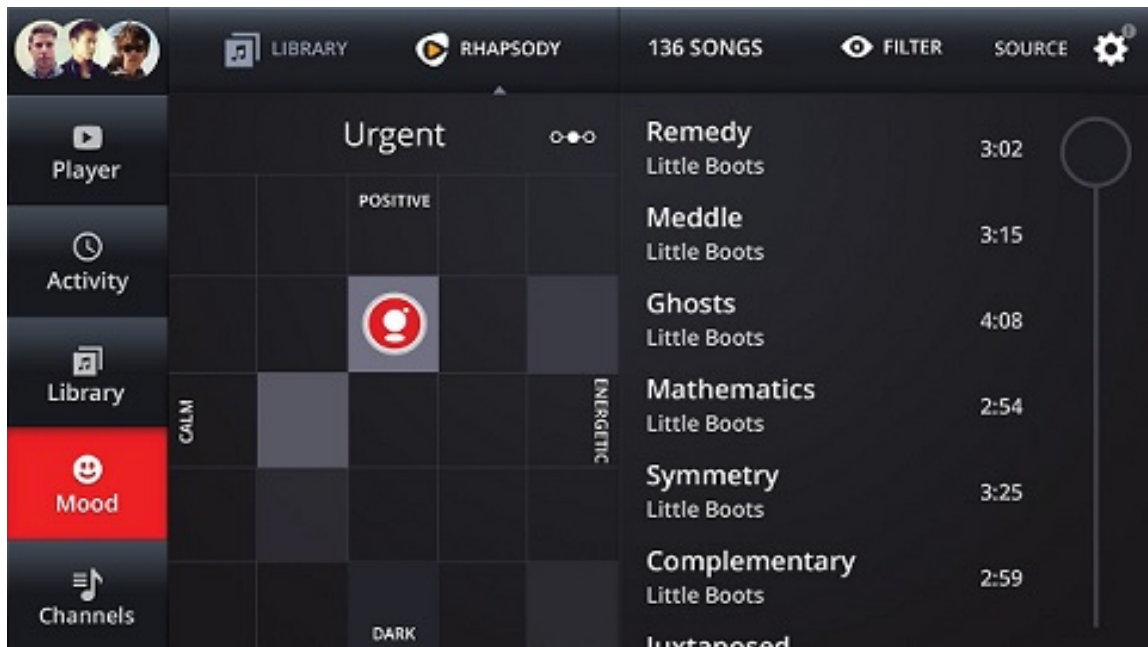
Slider Mood Navigation is shown below. Gracenote recommends this design because of its simplicity and ease-of-use in an automotive environment. It can support a touch-screen or scroll-wheel interface for selecting values. This design provides the ability to choose five discrete values on each dimension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



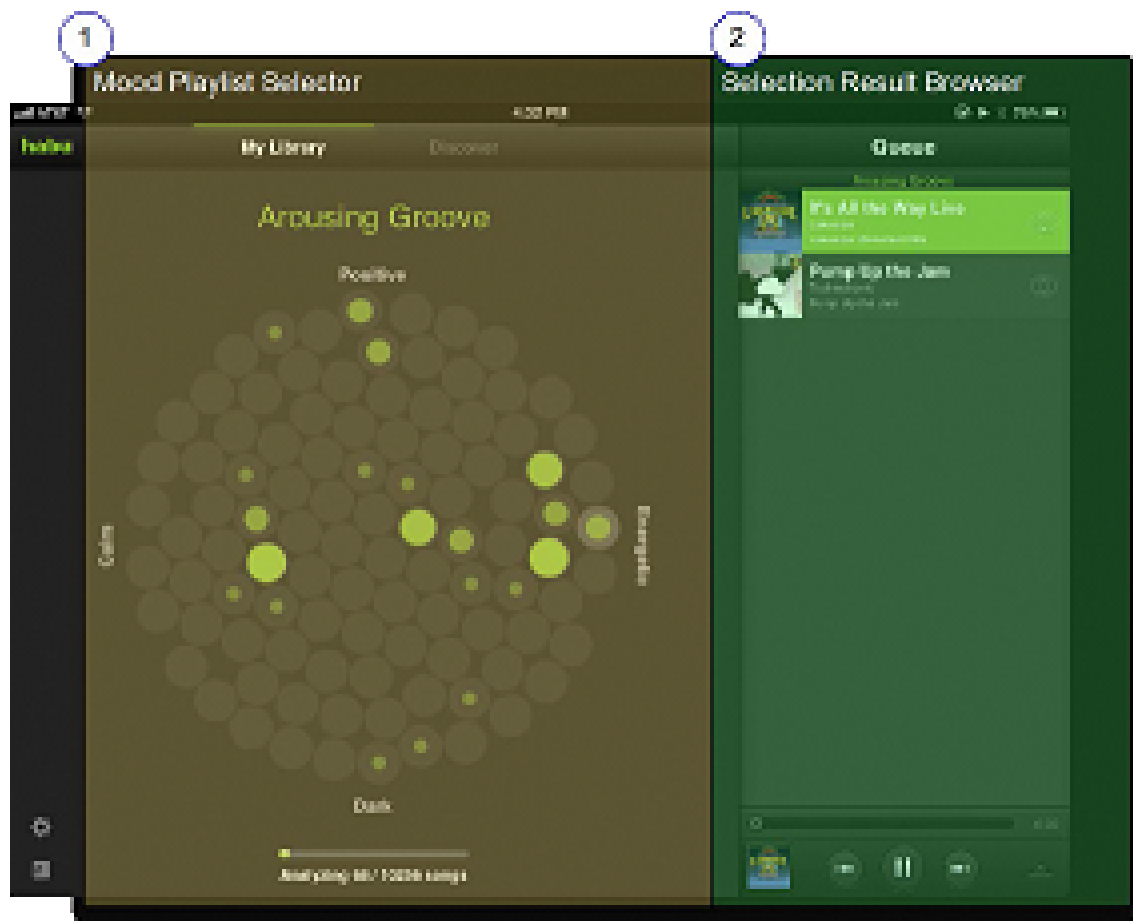
Bubble Magnitude Navigation

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

Mood	Valance (Emotion)	Arousal (Energy)
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
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
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

Mood	Valance (Emotion)	Arousal (Energy)
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
Soft Soulful	6	5
Dramatic Emotion	7	5
Strong / Stable	8	5
Soulful / Easygoing	9	5
Carefree Pop	10	5

Mood	Valance (Emotion)	Arousal (Energy)
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
Dark Pop Intensity	4	8
Energetic Abstract Groove	5	8
Fiery Groove	6	8

Mood	Valance (Emotion)	Arousal (Energy)
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

1.4.6 *Rhythm Overview*

Rhythm enables seamless integration of any audio source with online music services within a single application. Depending on the supported features of each online music service, an identified track can be used to link to a service to:

- Play that song
- Play more songs from that artist/album or
- Create an adaptive radio session

The identified track can be from any music source including terrestrial radio, CD, audio file or even another music service provided the track has been identified using Gracenote recognition technology.

Gracenote Rhythm allows you to create highly relevant and personalized adaptive radio stations and music recommendations for end users. Radio stations can be created with seed artists and tracks or a combination of genre, era, and mood, and can support Digital Millennium Copyright Act (DCMA) sequencing rules.

Rhythm includes the ability for an end user to specify whether they have played a track, like or dislike it, or want to skip past it. Rhythm can reconfigure the radio station playlist to more closely reflect end user preferences. In addition, it supports cover art retrieval, third-party links, and metadata content exploration.

Rhythm can:

- **Create a Radio Station:** Creating a radio station requires a User ID and a seed (artist, track, genre, era, or mood). Creating a radio station generates a track playlist.
- **Adapt to User Feedback:** End users can provide feedback (play, like, dislike, skip) about songs in the current radio station's playlist. This can result in modifying a playlist or generating a new one altogether.
- **Tune a Radio Station:** Reconfigure an existing radio station playlist based on track popularity and similarity.

- **Create a Playlist:** Rhythm can create a track playlist without the overhead of creating a radio station.

Radio station creation returns a radio station ID and a track playlist. The radio ID is used in subsequent actions on the station, its playlist, or its tuning parameters. Rhythm builds radio stations using GDOs.



Note: 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.

MusicID can retrieve the necessary GDOs by looking up artist names, album titles, or track titles. To create a playlist, Rhythm uses:

- Artist ID
- Track TUI (Title Unique Identifier)
- Genre, mood, or era attributes
- More than one of the above in combination

Multiple seed stations create a greater variety of results. Once you create a radio station, you can examine its entire playlist (up to 25 tracks at a time). End users can take various feedback event actions on the playlist such as:

- Track like
- Track dislike
- Track skipped
- Track played
- Artist like
- Artist dislike

When feedback is sent, Rhythm reconfigures the station's playlist accordingly. Rhythm supports the retrieval of the playlist at any time before or after feedback is provided.

A station's behavior may also be tuned by changing playlist generation behavior. For example, you can get difference responses based on options for popularity or similarity.

1.4.7 *MusicID-Match Overview*

MusicID-Match (also known as Scan and Match) uses a combination of waveform fingerprinting technologies to match tracks within an end-users collection to tracks within a cloud music provider's catalog, enabling instant playback from all devices without requiring upload.

Full-file fingerprinting technology is employed to identify the differences between varying length versions of the same track, and a highly robust fingerprint allows for the differentiation of highly similar tracks, such as remasters and clean/explicit versions.

MusicID-Match supports:

- Queries using MusicID-Match fingerprint types
- Query results management

A MusicID-File fingerprint is first generated for the purposes of candidate generation. Metadata is utilized to resolve or further refine the track candidate set.

A full-file fingerprint, using fingerprinting technology similar to that of MusicID Stream, is generated to perform any final disambiguation. All fingerprint matching is performed against a private data store containing only the required reference fingerprints for the customer's catalog.

MusicID-Match requires that all matching logic be implemented service-side.

1.5 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.

1.5.1 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

Please contact your Gracenote representative for more information.

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.

1.5.2 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.

1.5.2.1 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.

1.5.2.2 Artist Images

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

Video contributor images are most often provided in portrait orientation.

1.5.2.3 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.

1.5.2.4 Video Cover Art

Video cover art is most often taller than it is wide (portrait orientation). For most video cover art, this means that images will completely fill the vertical dimension of the requested image size, and will not fill the horizontal dimension. Therefore, while mostly fixed in height, video images may vary slightly in width. For example, requests for a "450" video image will likely return an image that is exactly 450 pixels tall, but close to 300 pixels wide.

As with CD cover art, the dimensions of video covers also include packaging variants such as box sets which sometimes result in significant variations in video image dimensions.

Video Image Dimension Variations

Video imagery commonly conforms to the shape of a tall rectangle with either a 3:4 or 6:9 aspect ratio. Image dimension characteristics of Gracenote Video imagery are provided in the following sections as guidelines for customers who want to implement Gracenote imagery into UI designs that rely on these aspect ratios. Gracenote recommends, however, that applications reserve square spaces in UI designs to accommodate natural variations in image dimensions.

Video (AV Work) Images

AV Work images typically conform to a 3:4 (width:height) aspect ratio.

3:4 (± 10%)	Narrower	Wider	Narrowest	Widest
98%	1%	1%	1:2	9:5

Video Product Images

Video Product images typically conform to a 3:4 (width:height) aspect ratio

3:4 (± 10%)	Narrower	Wider	Narrowest	Widest
90%	5%	5%	1:3	5:1

Video Contributor Images

Video Contributor images typically conform to a 6:9 (width:height) aspect ratio. Two ranges are provided due to larger variation in image dimensions.

6:9 (± 20%)	Narrower	Wider	Narrowest	Widest
90%	0%	10%	1:3	9:5

6:9 (± 10%)	Narrower	Wider	Narrowest	Widest
69%	1%	30%	1:3	9:5

1.5.3 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.

Gracernote 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 Gracernote images including changes to and support for existing image specifications.

1.5.3.1 Using a Default Image When Cover Art Is Missing

Occasionally, an Album result might have missing cover art. If the cover art is missing, Gracernote recommends trying to retrieve the artist image, and if that is not available, trying to retrieve the genre image. If none of these images are available, your application can use a default image as a substitute. Gracernote distributes an clef symbol image to serve as a default replacement. The images are in jpeg format and located in the /images folder of the package. The image names are:

- music_75x75.jpg
- music_170x170.jpg
- music_300x300.jpg

1.5.3.2 Image Resizing Guidelines

Gracernote 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. Gracernote 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, Gracernote recommends that applications use Gracernote images in their provided pixel dimensions without stretching or resizing.

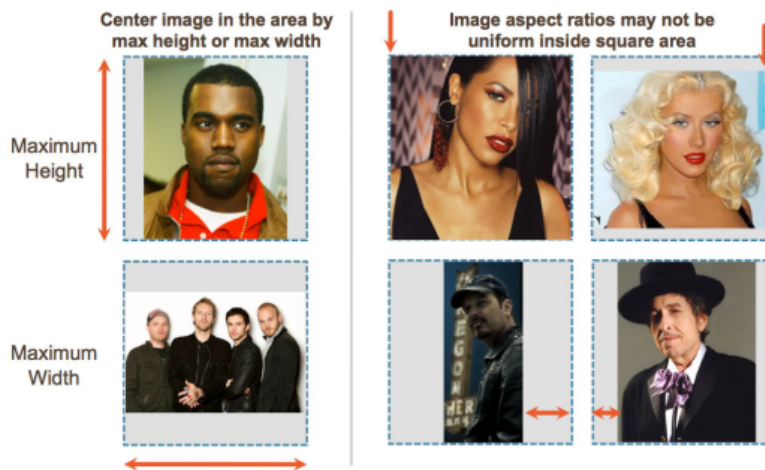
Gracernote resizes images based on the following guidelines:

- **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.



Note: 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.

Chapter 2 Setup and Samples

This section describes how to setup your development environment to create GNSDK applications. It also describes how to use sample applications to get you started with development.

2.1 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.

Module	Description
ACR module	Supports Automatic Content Recognition (ACR).
EPG module	Supports Electronic Programming Guide (EPG) features
FP Local module	Provides the option to store fingerprint data in local cache for faster local lookup.
Link module	A deprecated module that provides links to enriched metadata. Instead use the fetch APIs in Manager.
Manager module	Main module used by all applications. Other GNSDK libraries cannot function without it.

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.
DSP (Music Fingerprinting) 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. Also 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.
MusicID module	Provides for audio recognition using CD TOC-based search (MusicID CD), Text-based searches (MusicID Text), Fingerprint, and Identifier lookup functionality.
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.
MusicID Stream	Provides software 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.
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.

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.
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.

2.1.1 *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.

2.1.1.1 GNSDK Modules

- DSP
- LINK
- LOOKUP_LOCAL
- LOOKUP_LOCALSTREAM
- MANAGER
- MOOD
- MUSICID
- MUSICID_FILE
- MUSICID_MATCH
- MUSICID_STREAM
- PLAYLIST

- STORAGE_SQLITE
- SUBMIT

2.2 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" on page 86](#) for more information. This is not

required (unless you are doing local lookups), but highly recommended.

3. Perform queries to identify music. See *"Identifying Music" on page 104*
4. Parse query result objects for desired metadata. See *"Processing Returned Metadata Results" on page 158* for more information.
5. Exit the application when done, no cleanup necessary.

2.3 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](#).

2.3.1 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.



Note: 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.

2.3.1.1 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 alphanumeric characters separated with a hyphen. The numbers before the hyphen constitute the 'ID' and the alphanumeric characters after, the 'Tag'. A client ID/Tag string has the following format:

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

2.3.1.2 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.



Note: 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

2.3.2 *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.gracernote.gnsdk.*;
```

Objective-C

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

Windows Phone C#

```
using Gracernote;
```

C++

```
#include "gnsdk.hpp"
```

C#

```
using GracernoteSDK;
```

2.3.3 *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 Gracernote.

2.3.3.1 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

Java

```
// Load GNSDK native library
static {
    try {
        System.loadLibrary("gnsdk_java_marshall");
    } catch (UnsatisfiedLinkError unsatisfiedLinkError) {
        System.err.println("Native code library failed to load\n" +
            unsatisfiedLinkError.getMessage());
        System.exit(1);
    }
}

// Initialize GNSDK
gnsdk = new GnManager(libraryPath,           // SDK Libraries
location
                    gnsdkLicenseFilePath,    // License file path
and name
                    GnLicenseInputMode.kLicenseInputModeFilename); // Input
License as a file
```

Android Java

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

// Initialize GNSDK
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);
```

C++

```
/**
 * Initialize SDK
 */
GnManager gnMgr(licenseFile, kLicenseInputModeFilename);
```

C#

```
/* Initialize SDK */
GnManager manager = new GnManager(gnsdkLibraryPath, licenseFile,
GnLicenseInputMode.kLicenseInputModeFilename);
```

2.3.4 *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. Gracernote

uses the Client ID and Client Tag to verify that the licensed and allowable users quota has not been exceeded.

`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");
```

2.3.4.1 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`.



Note: 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.

Example implementation of `IGnUserStore` (C++):

```
/*-----
-----
*   class UserStore
*   Example implementation of interface: IGnUserStore
*   Loads and stores User data for the GnUser object. This sample
```

```
stores the
*   user data to a local file named 'user.txt'.
*   Your application should store the user data to an appropriate
location.
*/
class UserStore : public IGnUserStore
{
public:
    GnString
    LoadSerializedUser(gnsdk_cstr_t clientId)
    {
        std::fstream userRegFile;
        std::string fileName;
        std::string serialized;
        GnString userData;

        fileName = clientId;
        fileName += "_user.txt";

        userRegFile.open(fileName.c_str(), std::ios_base::in);
        if (!userRegFile.fail())
        {
            userRegFile >> serialized;
            userData = serialized.c_str();
        }
        return userData;
    }

    bool
    StoreSerializedUser(gnsdk_cstr_t clientId, gnsdk_cstr_t userData)
    {
        std::fstream userRegFile;
        std::string fileName;

        fileName = clientId;
        fileName += "_user.txt";

        /* store user data to file */
        userRegFile.open(fileName.c_str(), std::ios_base::out);
        if (!userRegFile.fail())
        {
            userRegFile << userData;
            return true;
        }
    }
}
```

```
        return false;
    }
};
```

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:

C++

```
UserStore userStore;
GnUser gnUser = GnUser(userStore, clientId, clientIdTag,
applicationVersion);
```

Java

```
GnUser gnUser = new GnUser( new UserStore(), clientId, clientIdTag,
CLIENT_APP_VERSION );
```

C#

```
GnUser gnUser = GetUser(manager, clientId, clientIdTag,
applicationVersion, lookupMode);
```

2.3.4.2 User Object Options

You can set/get the following options with a the `GnUser`'s `GnUserOptions` class:

- **CacheExpiration**—The maximum duration for which an item in the GNSDK query cache is valid.
- **DeviceModel**—Set the device model, used for runtime statistics.
- **LookupMode**—Options for online or local lookup. See the [Setting Local and Online Lookup Modes](#) topic for more information.
- **NetworkInterface**—Set a specific network interface to use with this object's connections. This can be useful for systems with multiple network interaces. Otherwise, the operating system determines the interface to use.

- **NetworkLoadBalance**—Enable/disable distributing queries across multiple Gracenote co-location facilities.
- **NetworkProxy**—Host name, username, and password for routing GNSDK queries through a proxy.
- **NetworkTimeout**—The network time-out for all GNSDK queries in milliseconds.
- **UserInfo**—Set information (location, manufacturer, OS, UID) about this user, used for runtime statistics.

Chapter 3 Develop and Implement

This section describes how to develop and implement GNSDK applications.

3.1 About this Documentation

This documentation 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-case (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.

3.2 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** - Granularity of returned Gracenote metadata, either Simplified or Detailed. Applies only to genres. If you want more, finer-grained genres, then use Detailed; otherwise, if storage or bandwidth need to be minimized and/or you can live with less granular genres, use Simplified.
- **Status callback (optional)** - One of the constructors takes a `GnStatusEventsListener` callback object

Once a locale is loaded it can be set as the group default. The GNSDK automatically associates certain returned response objects, allowing Gracenote descriptive data, such as genres, to be returned according to the locale's region and language.



Note: Locales have the following space requirements: 2MB for a music only locale, 6MB for a music and playlist locale. Loading a locale can be lengthy, so your application may want to perform this operation on a background thread to avoid stalling the main thread.



Note: 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).



Note: To cancel a Locale load at any time, set the "canceller" provided in any IGnStatusEvents delegate method. No cancel method is provided on GnLocale object because loading can happen on object construction.

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();
```

C++

```
/* Set locale with desired Group, Language, Region and Descriptor */
GnLocale locale( GnSDK::kLocaleGroupMusic,
    GnSDK::kLanguageEnglish,
    GnSDK::kRegionDefault,
```

```
GnSDK::kDescriptorSimplified,  
    user,  
    GNSDK_NULL); /* Use &statusEvents instead of GNSDK_NULL to  
view progress */  
  
/* Set this locale as default for the duration of gnsdk */  
locale.SetGroupDefault();
```

3.2.1 *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

3.2.2 *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.

3.2.3 *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.

3.2.3.1 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.

3.2.3.2 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" on page 86*.

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.



Note: 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" on page 94*.

Once your app receives a notification, it can choose to immediately do the update or do it later. Gracenote 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.



Note: 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.

3.2.4 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" on page 94*.

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.


Storage Provider Initialized	GnLookupMode Enum	Behavior
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.





Note: 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.

3.2.5 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.

Practice	Description
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>Note: 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>  Note: 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. </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>  Note: 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. </div>

3.3 Using Logging

The `GnLog` class has methods to enable Gracenote 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 Gracenote 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.



Note: Gracenote 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 Gracenote representative for more information.

3.3.1 Enabling GNSDK Logging

To use Gracenote 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.



Note: 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: 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);
```

C++

```
/* Enable GNSDK logging */  
GnLog sampleLog(  
    "sample.log", /* File to write log to (optional  
if using delegate) */  
    GnLogFilters().Error().Warning(), /* Include only error and  
warning entries */
```

```
GnLogColumns().All(),           /* Add all columns to log:
timestamps, thread IDs, etc */
GnLogOptions().MaxSize(0).Archive(false), /* Max size of
log: 0 means a new log file will be created each run. Archiving of
logs disabled. */
GNSDK_NULL                      /* Optional callback delegate for
logging messages */
);
sampleLog.Enable(kLogPackageAllGNSDK);
```

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.

3.3.2 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.

3.4 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 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.

3. **Results databases**—GNSDK applications can have dynamic databases called *Results databases* for persistent storage of either local or online search results and images. An application can have multiple Results databases open at one time, depending on application need.

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



Note: Gracernote databases are regularly updated. Your application is responsible for replacing old databases as new versions become available. This is typically accomplished by * shutting down GNSDK and replacing the database files.

3.4.1 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.



Note: * 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;
```

3.4.2 GNSDK Production 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.

3.4.3 GNSDK Databases

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

Database (<code>GnLookupLocal</code>)	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.

3.4.4 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

C++

```
/* Set default folder location for all SQLite storage */
storageSqlite.StorageLocation(".");
```

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);
```

3.4.5 Storing Results in Local Databases

You can save query results in local databases, which can allow your application to operate offline or increase performance. Note that local databases do not require that your application have a Gracenote-provided production database.



Note: Note that results databases do not support record deletion/expiration.

You can implement results databases using the `GnLookupLocal` and `GnLookupDatabase` classes:

To save query results in local results databases:

1. Enable local lookup and set its location (C++):

```
GnLookupLocal& gnLookupLocal = GnLookupLocal::Enable();
```

2. Allocate a `GnLookupDatabase` lookup provider with a database string identifier and a `GnConfig` object:

```
GnLookupDatabase OpenLocalDB(gnsdk_cstr_t dbId, gnsdk_cstr_t  
dbLocation, GnConfigOptionAccess accessMode)  
{  
    GnConfig config;  
  
    config.Set(kConfigOptionLocationLookupDatabaseAll, dbLocation);  
    // Set database location
```

```
    config.Set(kConfigOptionEnableLookupDatabaseMusicIDText, GNSDK_
VALUE_TRUE); // Enable Music ID text queries
    config.Set(kConfigOptionEnableLookupDatabaseMusicIDImages,
GNSDK_VALUE_TRUE); // Enable Music ID image queries
    config.Set(accessMode); // Set database access mode

    return GnLookupDatabase(dbId, config);
}
```

3. Store query results using **GnLookupDatabase methods** - **AddRecord** (album or contributor), and **AddImage**.

```
dbId.AddRecord(album) :
dbId.AddImage(image, size, asset);
```

Once results are stored, data will show up in local lookups.

3.4.6 *Getting Results 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.

3.4.6.1 *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.

3.4.6.2 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);
```

3.4.6.3 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.

3.4.7 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.

3.4.8 Managing Online Cache Size and Memory

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

- **FileSize**—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.

3.4.9 *Closing and Deleting a Results Database*

All databases are automatically closed when your application's `GnManager` object goes out of scope. You can also explicitly close a database with the `GnLookupDatabase`'s `Close` method. This method waits until any outstanding threads have finished accessing the database.

To delete a database, use the `GnLookupDatabase`'s `Delete` method.

3.4.10 *Managing Results Database Size*

You have the option to manage the Results database size. One simple solution is to delete and re-allocate once a specified has been reached or exceeded. For example (C++):

```
#define RESULT_DATABASE_MAX_SIZE    1048576 /* Maximum size for our
persistent result database */

....

/* Create a configuration object for use in opening our local lookup
database. */
config = new GnConfig();
```

```
/* Specify the location of the GDB files. */
config->Set(kConfigOptionLocationLookupDatabaseAll,
"..../..../..../sample_data/gdb");
/* Enable this database for MusicID Text queries. */
config->Set(kConfigOptionEnableLookupDatabaseMusicIDText, GNSDK_VALUE_
TRUE);
/* Enable this database for MusicID Images queries. */
config->Set(kConfigOptionEnableLookupDatabaseMusicIDImages, GNSDK_
VALUE_TRUE);
/* Set the access mode for the database */
config->Set(_access_mode);

/* Create Lookup Database */
p_GnLookupDB = new GnLookupDatabase("OurID", *config);

....

if (p_GnLookupDB->Size() > RESULT_DATABASE_MAX_SIZE)
{
    p_GnLookupDB->Close();
    GnLookupDatabase::Delete(*config);
    p_GnLookupDB = new GnLookupDatabase(id, *config);
}
```

3.5 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" on page 86* for more information.

3.5.1 Supported Lookup Modes

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

1. `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.
2. `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.
3. `kLookupModeOnlineCacheOnly`—Queries are done against the online cache only and does not perform a network lookup. The query fails if no online provider exists.
4. `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.
5. `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];
```

3.5.2 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`.

3.5.3 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];
```

3.5.4 Using Both Local and Online Lookup Modes

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

For example (C++):

```
/* Let's try a local lookup first */
music_id.Options().LookupMode(kLookupModeLocal);

/** Set the input text as album title, artist name, track title and
perform the query***/
GnResponseDataMatches response = music_id.FindMatches(
    "Nevermind",           // Album Title
    "Smells Like Teen Spirit", // Track Title
    "Nirvana",             // Artist Name
    GNSDK_NULL,
    GNSDK_NULL
);

/* See how many matches were found. */
gnsdk_uint32_t count = response.DataMatches().count();

if (count == 0)
{
    if (isNetworkAvailable())
    {
        cout << endl << "No results found locally, trying and online
query." << endl;
        /* Change the lookup mode on the query handle to 'Online No
Cache.'
        * Query caching is not used for automotive as it expires entries
too quickly.
        * Automotive use cases instead use the result DB to persist
results in an
        * embedded database.
        */
        music_id.Options().LookupMode(kLookupModeOnlineNoCache);

        response = music_id.FindMatches(
            "Nevermind",           // Album Title
```

```
        "Smells Like Teen Spirit",    // Track Title
        "Nirvana",                  // Artist Name
        GNSDK_NULL,
        GNSDK_NULL
    );
    // Check if we have a match
    count = response.DataMatches().count();
}
}
```

3.6 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 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.

3.6.1 *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;

    ingestester.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();
```

3.6.2 Designating a Storage Provider

A *storage provider* module implements GNSDK local storage. By default, no storage provider is enabled. To use SQLite, your application should instantiate a `GnStorageSqlite` object and call its `enable` method.

3.7 Identifying Music

GNSDK provides the following modules to identify music:

Music-ID—Using the `GnMusicId` class, you can identify music using:

- A CD TOC (table of contents)
- A text search
- An Audio Fingerprint
- A Gracenote identifier

See *"Identifying Music Using a CD-TOC, Text, or Fingerprints (MusicID)" on the facing page* for more information.

Music-ID File— Using the `GnMusicIdFile` class, you can identify music stored as audio files. See *"Identifying Audio Files (MusicID-File)" on page 123* for more information.

Music-ID Stream— Using the `GnMusicIdStream` class you can identify music from a streaming audio source. On some platforms, the `GnMic` class is available for listening to a device microphone. For more information, see [Identifying Streaming Audio](#).

3.7.1 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)" on page 123*.



Note: You must be licensed to use MusicID. Contact your Gracenote support representative with questions about product licensing and entitlement.

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.
- `GnMusicId` recognition events are performed synchronously, with the response object returned to your application.

Options for MusicID Queries

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

- **AudioSuitabilityProcessing**—Indicates whether `GnMusicId` should evaluate audio as being suitable for query processing. You can check suitability before making a query using a `GnMusicId` `infoGet` method call:

Java

```
GnString suitable = mid.infoGet  
(GnMusicIdInfo.kInfoSuitableForQuery); /* Returns 'true' or  
'false' */
```



Note: There are other options available that can give you finer control over defining, controlling, and assessing suitability. To implement this, you should talk to your Gracenote customer representative.

- **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`'s `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

```
NSString*   toc = @"150 14112 25007 41402 54705 69572 87335 98945
112902 131902 144055 157985 176900 189260 203342";

NSLog(@"\n*****MusicID TOC Query*****\n");

NSError *error = nil;
/*LookupStatusEvents statusEvents;*/
GnMusicId* musicid = [[GnMusicId alloc] initWithUser:user
statusEventsDelegate:[MusicIDLookupAlbumTOCEvent alloc]init]
error:nil];

[[musicid options] lookupData:kLookupDataContent bEnable:YES
error:&error];
GnResponseAlbums *response = [musicid findAlbumsWithCDTOC:toc
error:&error];
```

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`'s `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*.

Generating fingerprints is preferred when a device cannot immediately perform recognition, perhaps because it is temporarily disconnected from the Internet, and

wishes to do so later. Fingerprint data is much smaller than raw audio, putting less demand on storage resources.

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](#)

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 Cantametrix (CMX) fingerprint.

- **FingerprintEnd**—Finalizes fingerprint generation.

Identifying music using MusicID fingerprinting examples:

C++

```
/*-----  
-----  
 * GatherFingerprint  
 */  
virtual void  
GatherFingerprint(GnMusicIdFileInfo& fileInfo, gnsdk_uint32_t  
currentFile, gnsdk_uint32_t totalFiles, IGnCancellable& canceller)  
{  
    char    pcmAudio[2048] = {0};  
    gnsdk_bool_t complete    = GNSDK_FALSE;  
    gnsdk_cstr_t file;  
  
    GNSDK_UNUSED(currentFile);  
    GNSDK_UNUSED(totalFiles);  
    GNSDK_UNUSED(canceller);  
  
    try  
    {  
        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  
then 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";
}
else
{
    std::cout << "\n\nError: Failed to open input file: " << file
<< "\n\n";
}
catch (GnError& e)
{
    std::cout << e.ErrorAPI() << "\t" << std::hex << e.ErrorCode()
<< "\t" << e.ErrorDescription() << std::endl;
}
}
```

Java

```
@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
then 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 fingerprint.
        System.out.println("Warning: input file does not contain
        enough data to generate a fingerprint:\n" + audioFile.getPath());
    }

}

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

Objective-C

```
-(void) gatherFingerprint: (GnMusicIdFileInfo*)fileInfo currentFile:
(NSUInteger)currentFile totalFiles: (NSUInteger)totalFiles
cancellableDelegate: (id <GnCancellableDelegate>)canceller
{
    (void)currentFile;
    (void)totalFiles;
    (void)canceller;

    NSString* file = [fileInfo fileName:nil];
    bool complete = false;
```

```
if( file )
{
    NSFileHandle *fileHandle = [NSFileHandle
fileHandleForReadingAtPath:file];
    if( fileHandle != nil )
    {
        /* 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
then 44 bytes!
        */
        unsigned long long offset = 44;
        [fileHandle seekToFileOffset:offset];

        /* initialize the fingerprint
        * Note: The sample files are non-standard 11025 Hz 16-bit mono
to save on file size
        */
        [fileInfo fingerprintBegin:11025 audioSampleSize:16
audioChannels:1 error:nil];

        NSData* fileData = nil;
        do
        {
            fileData = [fileHandle readDataOfLength:2048];
            if( nil != [fileData bytes])
            {
                complete = [fileInfo fingerprintWrite:[fileData bytes]
audioDataSize:2048 error:nil];

                if( complete )
                {
                    break;
                }
            }
        }
        while ( nil != [fileData bytes] );

        if ( false == complete )
        {
            NSLog(@"Warning: input file does contain enough data to
generate a fingerprint:%@",file);
        }
    }
}
```

```
    }

    [fileInfo fingerprintEnd:nil];

    [fileHandle closeFile];
}
else
{
    NSLog(@"Error: Failed to open input file:%@",file);
}
}
}
```

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);

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

```

```
}  
}
```

Python

```
try:  
    wave_file = open(key_path, "rb")  
    # Skip past header. Note: the header may be different size on your  
    files.  
    wave_file.seek(44)  
    file_info.fingerprint_begin(11025, 16, 1)  
    pcm_audio = bytearray(wave_file.read(1024))  
    complete = False  
    while len(pcm_audio) and not complete:  
        pcm_data_buff = gnsdk.byte_buffer(len(pcm_audio))  
        for i, pcm_audio_byte in enumerate(pcm_audio):  
            pcm_data_buff[i] = pcm_audio_byte  
        complete = file_info.fingerprint_write(pcm_data_buff, len(pcm_  
audio))  
        pcm_audio = bytearray(wave_file.read(1024))  
        file_info.fingerprint_end()  
  
    except IOError as e:  
        print 'Exception reading audio file' + e.ErrorDescription()  
    except gnsdk.GnError as e:  
        print "Error in reading audio file: " + e.ErrorDescription()  
        exit(1)  
  
    if not complete :  
        print 'Warning: input file does contain enough data to generate  
a fingerprint:\n' , key_path
```

3.7.1.1 Best Practices for MusicID Text Match Queries

You can use the `GnMusicId` class' `FindAlbums` and `FindMatches` methods to identify music files in a collection based on one or more text values. This kind of query is called a *text match query*.

Possible text inputs are artist name, album title, and track title. You can specify these as fields in the query handle. The more fields you provide, the better and more accurate the result, so you should provide as many as possible.

The query response will contain either albums (with or without matched track data), or contributors (artists) if an album match was not available.



Note: Currently, online queries currently do not return contributors. Only albums are returned.

The following table shows the usable metadata returned for a given set of inputs.

Input			Use This Data
Album Title	Contributor/Artist Name	Track Title	
✓	✓	✓	Use all information: <ul style="list-style-type: none"> • Album • Album's primary artist • Matched track if available • Matched track's artist if available
✓	✓	✗	Use all available information: <ul style="list-style-type: none"> • Album • Album's primary artist

Input			Use This Data
Album Title	Contributor/Artist Name	Track Title	
✓	✗	✓	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
✗	✓	✓	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
✗	✓	✗	<ul style="list-style-type: none"> If the response is an album, only use the album's primary artist. If the response is an artist, use all available information.
✓	✗	✗	Do not send this query.
✗	✗	✓	Do not send this query.

3.7.1.2 Identifying Music in a Batch Query

The `GnMusicIdBatch` class gives you all the querying functionality and options contained in the `GnMusicId` class (see [Identifying Music Using a CD-TOC, Text, or Fingerprints](#)) for use in a batch process.

Batch processing allows you to make multiple queries in a single Gracenote Service request. A single batch request can contain different query types, and each query can have different inputs. For example, a batch lookup can have a

query for finding matches with text inputs in addition to finding matches based on fingerprint input.

Each batch query requires a unique string identifier. This lets your app associate a query with its results. Creating a batch query with an identifier already in use results in an error. Gracenote recommends batches between 20 and 50 queries.

In creating batch queries, please note the following:

- A batch containing 20 queries results in a measurable reduction in average query processing time as well as uplink and downlink transmission size without a significant increase to peak memory usage.
- A batch containing approximately 50 queries results in some reduction in average query processing time as well as uplink and downlink data transmissions, but the peak memory usage is higher.
- A batch greater than 50 queries often increases average query processing time as well as uplink and downlink data transmissions and requires a larger peak memory usage.
- Batch sizes of one are discouraged. It is more efficient to use a standard query if only one result is required.
- The options for a batch query or a non-batch query are the same (see the `GnMusicIdBatchOptions` class in the API reference).

The following C++ code shows how you can set up batch queries in text and add them to your `GnMusicIdBatch` object before calling the Gracenote service.



Note: Besides text, you can also query using a fingerprint, a TOC, or a GDO (Gracenote Data Object). For more information, see the `GnMusicIdBatch` class in the API reference or see the `musicid_lookup_batch_matches_text` sample program.

Batch Query Code Sample (C++)

```
/* Create batch query instance */  
GnMusicIdBatch musicIdBatch(user);
```

```
/* Create queries for batch processing. Every query must have a unique
identifier */
/* Track 1 */
GnMusicIdBatchQuery musicIdQuery1(musicIdBatch, "track_1");
musicIdQuery.SetText("Ultimate Jesse Cook (Disc 2)", "On Walks The
Night", "Jesse Cook", GNSDK_NULL, GNSDK_NULL);

/* Track 2 */
GnMusicIdBatchQuery musicIdQuery2(musicIdBatch, "track_2");
musicIdQuery.SetText("Join The Dots: B-Sides and Rarities 1978-2001
(Box Set CD 1)", "A Man Inside My Mouth", "The Cure", GNSDK_NULL,
GNSDK_NULL);

/* Additional queries can be created based on other inputs such as CD
TOC, fingerprint and identifier */

/* Batch query request to Gracenote Service */
musicIdBatch.FindMatches();

/*
   At this point all responses have been created with one call to the
database. You can now retrieve response using the unique query id.
*/
GnResponseDataMatches match_response;

/* Track 1 */
match_response = musicIdBatch.GetMatches("track_1");
/* process Track 1 response */

/* Track 2 */
match_response = musicIdBatch.GetMatches("track_2");
/* process Track 1 response */
```

3.7.2 *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. You can use the object to add more information about the audio file, such as textual metadata including track title and artist name, and a fingerprint if the audio file can be decoded to PCM.

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.
6. Call one of the `GnMusicIdFile` query methods.
7. Handle metadata results in an events delegate callback.

3.7.2.1 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



Note: 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 : MusicID-File 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

};
```

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 + " *\n" );

        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 + " *\n" );
    }

    @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 fingerprint.
            System.out.println("Warning: input file does not contain
            enough data to generate a 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 is not 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
```

```
}  
  
};
```

3.7.2.2 Adding Audio Files for Identification

To add audio files for identification:

1. Call the `GnMusicIdFile's 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's 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.



Note: 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).

Where online processing is enabled, portions of the audio file resolution into albums is performed within Gracenote Service rather than on the device. Your license must allow online processing.

An identification algorithm can be invoked multiple times on a single `GnMusicIdFile` instance, but it will only attempt to identify audio files added since the previous identification concluded. To re-recognize an entire collection ensure you use a new `GnMusicIdFile` object.

The identification process executes asynchronously in a worker thread and completed asynchronously. However, both synchronous and asynchronous identification methods are provided. Where synchronous identification is invoked, the identification is still performed asynchronously and results delivered via delegate implementing `IGnMusicIdFileEvents`, but the method does not return until identification is complete.

An identification process can be canceled. In this case the identification process stops and, if synchronous identification was invoked, the identify method returns.

Android Java

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

    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];
        }
    }
}
}
```

C++

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

Java

```
@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());
    }
}
```


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, IGnCancelable 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

```
/*-----
-----
*   GatherFingerprint
*/
-(void) gatherFingerprint: (GnMusicIdFileInfo*)fileInfo currentFile:
```

```
(NSUInteger)currentFile totalFiles: (NSUInteger)totalFiles
cancellableDelegate: (id <GnCancellableDelegate>)canceller
{
    (void)currentFile;
    (void)totalFiles;
    (void)canceller;

    NSString* file = [fileInfo getFileName:nil];
    BOOL complete = false;

    if( file )
    {
        NSFileHandle *fileHandle = [NSFileHandle
fileHandleForReadingAtPath:file];
        if( fileHandle != nil )
        {
            /* skip the wave header (first 44 bytes). the format of the s
files is known,
            * but please be aware that many wav file headers are larger
44 bytes!
            */
            unsigned long long offset = 44;
            [fileHandle seekToFileOffset:offset];

            /* initialize the fingerprinter
            * Note: The sample files are non-standard 11025 Hz 16-bit mo
save on file size
            */
            [fileInfo fingerprintBegin:11025 audioSampleSize:16 audioChar
error:nil];

            NSData* fileData = nil;
            do
            {
                fileData = [fileHandle readDataOfLength:2048];
                if( nil != [fileData bytes])
                {
                    complete = [fileInfo fingerprintWrite:fileData

                    if( complete )
                    {
                        break;
                    }
                }
            }
        }
    }
}
```

```
        }
        while ( nil != [fileData bytes] );

        if ( false == complete )
        {
            NSLog(@"Warning: input file does contain enough data
fingerprint:%@",file);
        }

        [fileInfo fingerprintEnd:nil];

        [fileHandle closeFile];
    }
    else
    {
        NSLog(@"Error: Failed to open input file:%@",file);
    }
}
```

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 then 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

```
@Override
```

```
public void
gatherFingerprint(GnMusicIdFileInfo fileInfo, long currentFile, long
totalFiles, IGnCancellable canceller) {
    boolean complete = false;
    try {
        File audioFile = new File(fileInfo.fileName());

        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 then 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 fingerprint. */
            System.out.println("Warning: input file does not
contain enough data to generate a 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());
        }
    }
}
```

3.7.2.3 Setting Options for MusicID-File Queries

To set an option for your MusicID-File query, instantiate a `GnMusicIdFileOptions` object using the `GnMusicIdFile's 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.

3.7.2.4 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: 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];
});
```

3.7.3 Identifying Streaming Music (MusicID Stream)

The functionality for identifying streaming music is contained in the MusicID 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. 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.



Note: 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.

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 and events delegate object. This establishes a MusicID-Stream audio channel.
3. Instantiate an audio source object representing the audio source you wish to identify. On some platforms, Gracenote provides the `GnMic` class, which is an `IGnAudioSource` implementation for the device microphone.
4. Call the `GnMusicIdStream` instance's `AudioProcessStart` method with your `IGnAudioSource` object. This starts the retrieval and processing of audio.



Note: 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.

5. To identify audio, call the `GnMusicIdStream`'s `IdentifyAlbumAsync` or `IdentifyAlbum` method. Results are delivered asynchronously as audio is received. The synchronous call blocks until enough audio is buffered in order to make an adequate identification attempt.
6. Handle results and statuses in your event delegate callbacks.

3.7.3.1 Implementation Notes

- At any point, your application can request an identification of buffered audio. The identification process spawns a thread and completes asynchronously. Two `GnMusicIdStream` identification methods are provided, one synchronous and one asynchronous (`IdentifySync` and `IdentifyAsync`). If `IdentifySync` is invoked, the identification is still performed asynchronously and the results delivered via a delegate. However, the method does not return until the identification is complete.
- 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. See *"Using a Local Fingerprint Database" on page 98* for more information. See *Using Local Audio Fingerprint Bundles* for more information.
- You can call `IdentifyCancel` to stop an identification operation. If synchronous identification had been invoked, the method returns. Note that canceling does not cease audio processing and your application can continue requesting identifications.
- 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.
- 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" on page 72* for more information.
- To aid in identification, you have the option to send broadcast data (Radio Data System (RDS) info, artist track/title, etc.) with your query using the `GnMusicIdStream`'s `text` method.

3.7.3.2 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:**

- **AudioSuitabilityProcessing**—Specifies whether `GnMusicIdStream` processes received audio is suitable for performing identification queries. If you enable this, the GNSDK will determine if the audio is good enough for a query. If not, an error is returned indicating that it is not.



Note: There are other options available that can give you finer control over defining, controlling, and assessing suitability. To implement this, contact your Gracenote customer representative.

- **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, or get, 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.
- **RevisionCheck**— Use this option if you have a full album object and you want to see if Gracenote has updated data for it.

3.7.3.3 Music-ID Stream Code Samples

Android Java

```
MusicIDStreamEvents musicIDStreamEvents = new MusicIDStreamEvents();

/**
 * GNSDK MusicID Stream event delegate
 */
private class MusicIDStreamEvents extends
GnMusicIdStreamEventsListener {

    HashMap<String, String> gnStatus_to_displayStatus;

    public MusicIDStreamEvents() {
        gnStatus_to_displayStatus = new HashMap<String,String>();
        gnStatus_to_displayStatus.put("kStatusStarted", "Identification
started");
        gnStatus_to_displayStatus.put("kStatusFpGenerated",
"Fingerprinting complete");

        //gnStatus_to_displayStatus.put
("kStatusIdentifyingOnlineQueryStarted", "Online query started");
        gnStatus_to_displayStatus.put("kStatusIdentifyingEnded",
"Identification complete");
    }

    @Override
    public void statusEvent( GnStatus status, long percentComplete, long
bytesTotalSent, long bytesTotalReceived, IGnCancellable cancellable )
    {
        //setStatus( String.format("%d%",percentComplete), true );
    }

    @Override
```

```
public void musicIdStreamStatusEvent( GnMusicIdStreamStatus status,
IGnCancellable cancellable ) {
    if(gnStatus_to_displayStatus.containsKey(status.toString())){
        setStatus( String.format("%s", gnStatus_to_displayStatus.get
(status.toString())), true );
    }
}

@Override
public void musicIdStreamResultAvailable( GnResponseAlbums result,
IGnCancellable cancellable ) {
    activity.runOnUiThread(new UpdateResultsRunnable( result ));
    setStatus( "Success", true );
    setUIState( UIState.READY );
}
}

private void startContinuousListening() throws GnException{

    if ( gnMicrophone == null ){
        gnMicrophone = new GnMic();
        gnMicrophone.start();
    } else {
        gnMicrophone.resume();
    }

    GnMusicIdStream = new GnMusicIdStream( gnUser, musicIDStreamEvents
);
    queryObjects.add( GnMusicIdStream.canceller() ); // retain event
object so we can cancel if requested
    GnMusicIdStream.options().resultSingle( true );

    Thread audioProcessThread = new Thread(new AudioProcessRunnable());
    audioProcessThread.start();
}

/**
 * GnMusicIdStream object processes audio read directly from GnMic
object
 */
class AudioProcessRunnable implements Runnable {

    @Override
    public void run() {
        try {
```

```
GnMusicIDStream.audioProcessStart( gnMicrophone );

} catch (GnException e) {
    Log.e( appString, e.getErrorCode() + ", " +
e.getErrorDescription() + ", " + e.getErrorModule() );
    showError( e.getErrorAPI() + ": " + e.getErrorDescription() );
}
}
}
```

Objective-C

```
-(void) setupMusicIDStream
{
    __block NSError *musicIDStreamError = nil;
    self.gnMusicIDStream = [[GnMusicIDStream alloc]
initWithUser:self.gnUser musicIDStreamEventsDelegate:self
error:&musicIDStreamError];

    musicIDStreamError = nil;
    [self.gnMusicIDStream optionResultSingle:YES
error:&musicIDStreamError];

    musicIDStreamError = nil;
    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_
DEFAULT, 0), ^
    {
        [self.gnMusicIDStream
audioProcessStartWithAudioSource:self.gnMic
error:&musicIDStreamError];

        if(musicIDStreamError)
        {
            dispatch_async(dispatch_get_main_queue(), ^
            {
                NSLog(@"Error while starting Audio Process With AudioSource
- %@", [musicIDStreamError localizedDescription]);
            });
        }

    });
}

#pragma mark - GnMusicIDStreamEventsDelegate Methods
```



```
-(void) midstreamStatusEvent:(GnMusicIDStream*) gnMusicIDStream
status:(GnAudioStatus) status
{
    NSString *statusString = nil;

    switch (status)
    {
        case AudioStatusInvalid:
            statusString = @"Error";
            [self performSelectorOnMainThread:@selector(updateStatus:)
withObject:statusString waitUntilDone:NO];

            break;
        case AudioStatusIdentifyingStarted:
            statusString = @"Identifying";
            [self performSelectorOnMainThread:@selector(updateStatus:)
withObject:statusString waitUntilDone:NO];
            break;

        /*... More audio statuses */

    }
}
```

Windows Phone C#

```
// Create and setup gracenote musicid stream instance
App.mMusicIDStream = new GnMusicIdStream(App.mGnUser, this);
App.mMusicIDStream.Options.ResultSingle(true);
App.mMusicIDStream.Options.LookupData
(GnLookupData.kLookupDataContent, true);
App.mDispatcherTimer.Start();
App.mMicrophone.Start();
App.mMusicIDStream.AudioProcessStart((uint)App.mMicrophone.SampleRate,
16, 1);

#region IGnMusicIdStreamEvents

void IGnMusicIdStreamEvents.MusicIdStreamResultAvailable
(GnResponseAlbums result, IGnCancellable canceller)
{
    ClearResults();
    ShowAlbums(result, true, false);
}
```

```
}

void IGnMusicIdStreamEvents.MusicIdStreamStatusEvent
(GnMusicIdStreamStatus status_, IGnCancellable canceller)
{
    return;
}

void IGnMusicIdStreamEvents.StatusEvent(GnStatus status_, uint
percent_complete, uint bytes_total_sent, uint bytes_total_received,
IGnCancellable canceller)
{
    Deployment.Current.Dispatcher.BeginInvoke(() =>
    {
        string status = null;
        switch (status_)
        {
            case GnStatus.kStatusDisconnected: status = "Disconnected";
break;
            case GnStatus.kStatusBegin:      status = "Begin";          break;
            case GnStatus.kStatusComplete:   status = "Complete";       break;
            case GnStatus.kStatusConnecting: status = "Connecting";      break;
            case GnStatus.kStatusProgress:   status = "Progress";       break;
            case GnStatus.kStatusReading:    status = "Reading";        break;
            case GnStatus.kStatusSending:    status = "Sending";        break;
            case GnStatus.kStatusUnknown:    status = "Unknown";        break;
            case GnStatus.kStatusWriting:    status = "Writing";        break;
            case GnStatus.kStatusErrorInfo:  status = "ErrorInfo";      break;
        }

        TBStatus.Text = "Status : " + status + "\t(" + percent_
complete.ToString() + "%)";
    });
}
#endregion
```

C++

```
/*-----
-----
* do_musicid_stream
*/
static void
do_musicid_stream(GnUser& user)
```

```
{
    std::cout << std::endl << "*****Sample MID-Stream Query*****" <<
    std::endl;

    MusicIDStreamEvents* mids_events = new MusicIDStreamEvents;

    try
    {
        GnMusicIdStream mids(user, mids_events);

        /* create microphone to use as audio source */
        GnMic    mic(44100, 16, 1);
        /* pass Mic through GnWavCapture to record audio for testing
        (optional) */
        GnWavCapture wavrec(mic, "mic_record.wav");

        /* We want to add Content (eg: image URLs) data to our lookups */
        mids.Options().LookupData(kLookupDataContent, true);

        /* starts identification in background thread - will wait for
        audio if none yet */
        mids.IdentifyAsync();

        /* provide audio continuously until
        GnMusicIdStream::AudioProcessStop() is called */
        mids.AudioProcessStart(wavrec);

        /* We called GnMusicIdStream::AudioProcessStop() on 'gnsdk_mids_
        identifying_ended'
        * status callback which caused the above to return */
    }
    catch (GnError& e)
    {
        std::cout << e.ErrorAPI() << "\t" << std::hex << e.ErrorCode() <<
        "\t" >> e.ErrorDescription() << std::endl;
    }

    delete mids_events;
}

/* IGnStatusEvents : overrider methods of this class to get delegate
callbacks */
class MusicIDStreamEvents : public IGnMusicIdStreamEvents
{
    /*-----
```

```
-----
*   StatusEvent
*/
void
    StatusEvent(gracenote::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;

        //...more statuses
    }
}

/*-----
-----
*   MidStreamStatusEvent
*/
virtual void
    MusicIdStreamStatusEvent(GnMusicIdStreamStatus status,
IGnCancellable& canceller)
{
    switch (status)
    {
        case gnsdk_mids_status_invalid:
            std::cout <<"Invalid status!" << std::endl;
            break;

        case gnsdk_mids_identifying_started:
            std::cout <<"Identification: started" << std::endl;
            break;

        //...more statuses
    }
}
```

```
    }

    /*-----
    -----
    *   MidStreamResultAvailable
    */
    virtual void
        MusicIdStreamResultAvailable(GnResponseAlbums& result,
        IGnCancellable& canceller)
    {
        if (result.ResultCount() > 0)
        {
            metadata::album_iterator it_album = result.Albums().begin();

            for (; it_album != result.Albums().end(); ++it_album)
            {
                GnAlbum album = *it_album;

                /* Get Track Artist, if not available, use Album Artist */
                gnsdk_cstr_t artist_name = album.TracksMatched()[0]->Artist
                ().Name().Display();
                if (artist_name == GNSDK_NULL)
                    artist_name = album.Artist().Name().Display();

                /* Get cover art URL for 'small' image */
                gnsdk_cstr_t cover_url = album.Content
                (kContentTypeImageCover).Asset(kImageSizeSmall).Url();
                if (cover_url == GNSDK_NULL)
                    cover_url = "no url";

                std::cout <lt; std::endl;
                std::cout <lt; "      Track : " <lt; album.TracksMatched()[0]-
                >Title().Display() <lt; std::endl;
                std::cout <lt; "      Artist: " <lt; artist_name <lt; std::endl;
                std::cout <lt; "      Cover : " <lt; cover_url <lt; std::endl;
            }
        }
        else
        {
            std::cout <lt; "      No match found" <lt; std::endl;
        }
        std::cout <lt; std::endl;

        GNSDK_UNUSED(canceller);
    }
}
```

```
};
```

3.8 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.

3.8.1 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];
    }
    // ...
}
```

3.8.2 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: 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.

3.8.3 *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" on page 72*.

3.8.4 *Accessing Enriched Content using Asset Fetch*

To access enriched metadata content, such as cover art and artist images, you can purchase additional metadata entitlements, and then use the Manager Asset Fetch APIs to get enriched content from response objects (`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.

3.8.4.1 *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];
```

3.8.4.2 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.

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 Gracernote service.

3.8.4.3 Retrieving a Content Asset

You can use the `GnAssetFetch` class to access an asset from local storage, the Gracernote service, or any Internet location with a `GnAsset` `urlHttp` field or `urlHttps` field or `urlLocal` field (for local fetches) and retrieve its content as raw byte data. Note that the `GnAsset` class has a `url` field, but that has been deprecated.

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.

Retrieving Local or Online Content

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. You can also set local or online lookup mode when you instantiate a `GnAssetFetch` object with a `lookupMode` parameter:

C++

```
GnAssetFetch  gnAssetFetch(user, url, kLookupModeLocal,  
eventHandler);
```



Note: Retrieving an asset online could result in some network delay depending on asset size.

If your lookup mode is set for online and the provided URL is local, you will get an unsupported functionality error.

3.8.4.4 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).urlHttp();
        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 (IOException e) {
                // Handle exception
            }
        }
        imageView.setImageDrawable(coverArt);
    }
}
```

```
        } 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 urlhttp]];
    GnContent *artistImageContent = nil;//[album
content:kContentTypeImageArtist];
    GnAsset *artistImageAsset = [artistImageContent
asset:kImageSizeSmall];
    NSString *artistImageURLString = [NSString
```

```
stringWithFormat:@"http://%@", [artistImageAsset urlHttp]];
    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).UrlHttp;
    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.UrlHttp());
            }
        }
    }
}
```

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/dk  
cbDgrOqBMIQJZMmvysjCkx10ppXc68ZcgU0SgLelyjfolTt7Ix/cn32BvcbeuPkAk0WwwR  
eVdcSLu08cYxAGcQrEE+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() );
}
}
```

3.8.5 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.



Note: Your application must be licensed to access this metadata. Contact Gracenote for more information.

C++

```
/*-----  
-----  
 *  navigate_audio_work  
 */  
static void  
navigate_audio_work(GnAudioWork gnAudioWork, gnsdk_uint32_t tab_index)  
{  
    gnsdk_char_t tab_string[1024];  
  
    create_tab_string(tab_index, tab_string);  
    tab_index += 1;  
  
    printf("%sAudio Work:\n", tab_string);  
  
    display_value("Tui", gnAudioWork.Tui(), tab_index);  
  
    /* Navigate Title Official from Audio Work */  
    navigate_title_official(gnAudioWork.Title(), tab_index);  
  
    display_value("Composition Form", gnAudioWork.CompositionForm(),  
tab_index);  
  
    return;  
}  
  
/*-----  
-----  
 *  navigate_track  
 */  
static void  
navigate_track(GnTrack track, gnsdk_uint32_t tab_index)  
{  
    gnsdk_char_t tab_string[1024];  
  
    create_tab_string(tab_index, tab_string);  
    tab_index += 1;
```

```
printf("%sTrack:\n", tab_string);

display_value("Track Tui", track.Tui(), tab_index);

display_value("Track Number", track.TrackNumber(), tab_index);

/* Navigate credit from track artist */

GnArtist artist = track.Artist();
if (artist.native())
{
    navigate_credit(track.Artist(), tab_index);
}

/* Navigate Title Official from track */
navigate_title_official(track.Title(), tab_index);

display_value("Year", track.Year(), tab_index);

/*Gnere levels from track*/
display_value("Genre Level 1", track.Genre(kDataLevel_1), tab_index);
display_value("Genre Level 2", track.Genre(kDataLevel_2), tab_index);
display_value("Genre Level 3", track.Genre(kDataLevel_3), tab_index);

/* Navigate Audio Works from track */
metadata::audio_work_iterator it_audio_work = track.AudioWorks().begin();
for (; it_audio_work != track.AudioWorks().end(); ++it_audio_work)
{
    navigate_audio_work(*it_audio_work, tab_index);
}
}
```

3.8.5.1 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>

```

```
</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>
    <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>
```

3.9 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. The GNSDK supports multiple user collections and therefore multiple Collection Summaries.

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.

3.9.1 *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.

3.9.2 *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`'s `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);
```

3.9.2.1 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);
```

3.9.3 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.



Note: In some cases an audio track may not contain enough information to match it with a single `GnTrack` object, in such caes you application can use the available information to add it to the Collection Summary. For example if only album title or artist name information is available `GnMusicId` could be used to match a `GnAlbum` or `GnContributor` object which can then be added. Similarly if only genre information is available the Lists's subsystem could be used tomatch a `GnListElement` object which can then be added.

3.9.4 Working with Local Storage

Playlist Collection Summaries are stored entirely in heap memory. To avoid re-creating them every time your application starts you should store them persistently. 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.** Alternatively, a Collection Summary can be serialized into a string that your application can store persistently and later use to reconstitute a Collection Summary in memory.

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:

- **Location**—Setting a storage location specifically for playlist collections
- **Remove**—Removing a collection from storage
- **Load**—Loading a collection from storage
- **Names**—Getting an iterator for names of stored collections
- **Compact**—Compacting storage

3.9.5 *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.



Note: Note that a "More Like This" playlist will not include the seed track. If you want the seed track, you will need to add it to the playlist yourself.

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. Default is 25.
<code>MaxPerArtist</code>	Maximum number of tracks per artist returned in a 'More Like This' playlist. Must evaluate to a number greater than 0. Default is 2.
<code>MaxPerAlbum</code>	Maximum number of results per album returned. Must evaluate to a number greater than 0. Default is 1.
<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. Default is 0.

C++

A More Like This playlist can be generated from a seed, which can be any Gracenote data object. For example, you can use `GnMusicId` to perform a text search for a specific track and receive a `GnAlbum` object. Your application can provide the `GnAlbum` object as the seed.



Note: Do not use the `GnTrack` object as the seed, Playlist will use the "matched" `GnTrack` object to determine which track on the album to use.

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.kMoreLikeThisMaxPerAl
bum, 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 ) );
```

3.9.6 *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.

3.9.7 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());  
}
```

3.9.8 *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.



Note: 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.

3.9.8.1 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.

3.9.9 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.

To synchronize a collection summary to physical media:

1. **Iterate the physical media** – Add all existing (current and new) unique identifiers, using the `GnPlaylistCollection's SyncProcessAdd` method. This can be done in a loop.
2. **Process the collection** – Call 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.

3.9.9.1 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.

3.9.9.2 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 identifiers or delete missing identifiers from the Collection Summary. For more information on adding items, see [Populating a Collection Summary](#). To remove items, use the `GnPlaylistCollection's Remove` method.

3.9.10 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.

Mood terminology

- **Mood Provider**—A collection of audio tracks such as a Playlist Collection Summary (`GnPlaylistCollectionSummary`)
- **Mood Presentation**—An instance of `GnMoodgridPresentation` representing a 5x5 or 10x10 grid containing

To implement Mood:

1. Allocate a `GnMoodgrid` class.
2. Enumerate the data sources using Mood Providers
3. Create and populate a Mood Presentation
4. Filtering the results, if needed

3.9.10.1 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 and DSP 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.

3.9.10.2 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

3.9.10.3 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);
```

3.9.10.4 Iterating Through a Mood Presentation

Each cell of the Presentation is populated with a mood name an 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());  
  
    }  
}
```

3.9.10.5 Filtering Mood Results

You can use genre, origin, and era to filter Mood results. To do this, use the `GnMoodgridPresentation`'s `addFilter` method. 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 punk rock genre.

Java:

```
myPresentation.addFilter("pres1", kMoodgridListTypeGenre, "punk",  
kConditionTypeInclude);
```

3.9.11 Playlist PDL Specification

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 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.

3.9.11.1 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

Keyword	Description	Required or Optional	PDL Statement Order
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.</p> <p>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 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.

3.9.11.2 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"
```

3.10 Implementing Rhythm

You can use the Rhythm API to:

- Create radio stations with a *seed* value.
- Generate a playlist with or without having to create a radio station.
- Adjust a radio station's playlist with feedback events and tuning parameters

Using Rhythm in your application involves doing some or all of the following:

1. Creating a Rhythm query with initial seed value.
2. Set Rhythm query options to fine-tune the results you want to see returned.

3. Generating a playlist, either as recommendations or when creating a radio station.
4. Retrieving the playlist.
5. For radio stations:
 - Sending feedback events (track/artist like/dislike/skip and so on.) which may alter the station's playlist queue.
 - Retrieving playlists altered from feedback events
 - Saving a radio station for later retrieval.

3.10.1 *Creating a Rhythm Query*

Before using Rhythm, follow the usual steps to initialize GNSDK. The following GNSDK modules must be initialized:

- GNSDK Manager
- MusicID (optional to retrieve GnDataObjects)
- Playlist (optional to use generated playlists)
- Rhythm

For more information on initializing the GNSDK modules, see “Initializing an Application.”

To create a Rhythm query:

1. Instantiate a `GnRhythmQuery` object with your User object.
2. Call the `GnRhythmQuery`'s `AddSeed` method with a `GnDataObject` seed that you can get with a MusicID query.



Note: If you are using MusicID to recognize music, you must enable Playlist and DSP data in your MusicID query.

Code Samples

C++



```
// Instantiate the query with a Gracenote user object
GnRhythmQuery rhythmQuery(user);

// Assume we start with a serialized data object (from MusicID)
// to create our seed.
GnDataObject seed = GnDataObject::Deserialize(serializedDataObject);
rhythmQuery.AddSeed(seed);
```

Java

```
// Instantiate the query with a Gracenote User object
GnRhythmQuery rhythmQuery = new GnRhythmQuery(user);
// Assume we start with a serialized data object (from MusicID)
// to create our seed.
GnDataObject seed = GnDataObject.deserialize(serializedDataObject);
rhythmQuery.addSeed(seed);
```

3.10.2 Setting Rhythm Query Options

You can use the following `GnRhythmQueryOptions` methods to set options for a Rhythm Query.

- **FocusPopularity**—Set with a number indicating how popular tracks should be. Range is 0-1000. Default is 1000 (very popular).
- **FocusSimilarity**—Set with a number indicating how similar tracks should be. Range is 0-1000. Default is 1000 (very similar).
- **LookupData**—Request additional content in your results: classical, sonic, external IDs (3rd-party) identifiers, global IDs, credits, and so on. See the `GnLookupData` enums for a complete list.
- **RecommendationMaxTracksPerArtist**—Specify a maximum number of tracks per artist for recommended playlist results. Range is 1-1000.
- **RecommendationRotationRadio**—Specifying `true` will cause results to be sequenced in a radio-like fashion. Choosing `false` will return normal recommendation results without sequencing. Default is `false`.
- **ReturnCount**—Specify how many tracks can be returned. The range is 1-25. Default is 5.
- **stationDMCA**—DMCA stands for Digital Millenium Copyright Act. When creating a radio station, you have the option to enable DMCA rules, which

reduces the repetition of songs and albums in conformance with DMCA guidelines.

3.10.3 *Generating Recommendations*

Once you have a seeded query, you can create a recommendations playlist, which uses less overhead than also creating a radio station

To generate a Rhythm query recommendations playlist:

1. Call your `GnRhythmQuery` object's `GenerateRecommendations` method.

C++

```
GnResponseAlbums recommendations =  
rhythmQuery.GenerateRecommendations();
```

Java

```
GnResponseAlbums recommendations =  
rhythmQuery.generateRecommendations();
```

2. Iterate through the generated `GnResponseAlbums` object.

Once you have a playlist, you can iterate through it to get album and track information.

3.10.4 *Creating a Radio Station and Playlist*

Creating a radio station and a playlist allows you to modify the playlist through feedback or tuning.

To create a radio station and playlist:

1. Instantiate a `GnRhythmStation` object, using your `GnRhythmQuery` object:

C++

```
GnRhythmStation rhythmStation(rhythmQuery);
```

Java

```
GnRhythmStation rhythmStation = new GnRhythmStation(rhythmQuery);
```

The Rhythm station object is how you provide feedback and tuning information.

2. To generate a radio station playlist, use the `GnRhythmStation GeneratePlaylist` method:

C++

```
GnResponseAlbums playlist = rhythmStation.GeneratePlaylist();
```

Java

```
GnResponseAlbums playlist = rhythmStation.generatePlaylist();
```

3.10.5 Providing Feedback

Once you have created a station, you cannot generate a new playlist for it with a different seed. To use a different seed you would have to create a new station. However, you can use feedback events to modify an existing station's playlist. The following `GnRhythmEvent` enums indicate the supported feedback events and how they affect a radio station's playlist:

- `kRhythmEventTrackPlayed`— Track marked as played. Advances the play queue (drops track being played and adds additional track to end of queue).
- `kRhythmEventTrackSkipped`— Track marked as skipped. Advances the queue by one.
- `kRhythmEventTrackLike`— Track marked as liked. Does not change the playlist queue.
- `kRhythmEventTrackDislike`— Track marked as disliked. Modifies the playlist queue but does not drop the disliked track. To do that, combine this with a track skipped event.

- `kRhythmEventArtistLike`— Artist marked as liked. Does not change the playlist queue.
- `kRhythmEventArtistDislike`— Artist marked as disliked. Modifies the playlist queue.

For a more detailed explanation on how feedback events modify a playlist, reference the Rhythm API documentation on the Gracenote developer portal: <https://developer.gracenote.com/sites/default/files/web/rhythm/index.htm>

See the *How Feedback Events Affect the Queue* article.

To provide feedback about a Station's playlist use the Station object's `Event` method. This method takes a `GnRhythmEvent` enum and a `GnDataObject`. You set the event, and either the track or artist for the event. For example, the following two functions send event information about a track or an artist to a station.

Code samples

C++

```
GnArtist artist = playlist.Albums().at(1)->Artist();
rhythmStation.Event(kRhythmEventArtistDislike, artist);

GnTrack track = playlist.Albums().at(1)->TrackMatched();
rhythmStation.Event(kRhythmEventTrackLike, track);
```

Java

```
GnArtist artist = playlist.albums().at(1).next().artist();
rhythmStation.event(kRhythmEventArtistDislike, artist);

GnTrack track = playlist.albums().at(1).next().trackMatched();
rhythmStation.event(kRhythmEventTrackLike, track);
```

3.10.6 Tuning a Radio Station

You can set options to adjust a radio station and its playlist after initial creation. Use the Rhythm station object's `Options()` methods to change values. These

options are the same as the Rhythm query options.

For example, the following C++ code turns DMCA support on:

```
rhythmStation.Options().StationDMCA(true);
```

3.10.7 *Saving and Retrieving Radio Stations*

Rhythm saves all created radio stations within the Gracenote service. Stations can be recalled through the station ID, which your application can store locally. To retrieve the station ID for storage, use the Station object's `StationId()` method.

Once you have the station ID, an application can save it to persistent storage and recall it at any later date. Station IDs are permanently valid. To retrieve a station create a `GnRhythmStation` object with the Station ID and associated user. This retrieval can take place during your app's current running or any future running.

For example, the following C++ code saves and retrieves a Station:

```
// Get station ID
gnsdk_cstr_t stationId = rhythmStation.StationId();

// Save the ID for later retrieval...

// Create a station based on the saved station ID
GnRhythmStation rhythmStation(stationId, user);
```

3.11 Best Practices and Design Requirements

3.11.1 *Managing Image Dimensions*

Gracenote music and video 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 music and video images including changes to and support for existing image specifications.

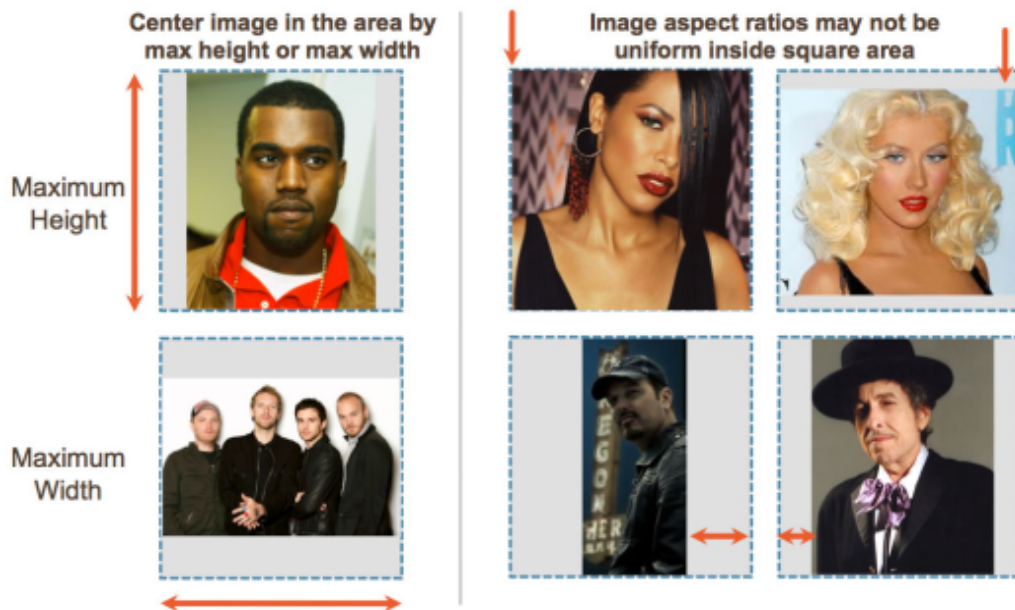
3.11.1.1 Image Resizing Guidelines

Gracenote music and video 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.



Note: 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.

3.11.1.2 Image Formats and Dimensions

Applications must specify an images size when requesting an image. Gracenote provides images in that fit within six square dimensions, as shown in the following table. For backward compatibility, Gracenote also supports legacy video image dimensions.

SIZE="NAME"	Current Support		Legacy Support	
	Width	Height	Width	Height
THUMBNAIL	75	75	40	<= 60
SMALL	170	170	160	<= 240
300	300	300		
MEDIUM	450	450	300	<= 450
LARGE	720	720	480	<= 720
XLARGE	1080	1080	720	<= 1080



Note: 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.

3.11.1.3 Common Media Image Dimensions

Music, video and artist 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

While 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.

Video Cover Art

Video cover art is most often taller than it is wide (portrait orientation). For most video cover art, this means that images will completely fill the vertical dimension of the requested image size, and will not fill the horizontal dimension. Therefore, while mostly fixed in height, video images may vary slightly in width. For example,

requests for a "450" video image will likely return an image that is exactly 450 pixels tall, but close to 300 pixels wide.

As with CD cover art, the dimensions of video covers also include packaging variants such as box sets which sometimes result in significant variations in video image dimensions.

Artist Imagery

Artist and Contributor images, such as publicity photos, come in a wide range of sizes and both portrait and landscape orientations. Video contributor images are most often provided in portrait orientation.

3.11.1.4 Variations in Video Image Dimensions

Video imagery commonly conforms to the shape of a tall rectangle with either a 3:4 or 6:9 aspect ratio. Image dimension characteristics of Gracenote Video imagery are provided in the following sections as guidelines for customers who want to implement Gracenote imagery into UI designs that rely on these aspect ratios. Gracenote recommends, however, that applications reserve square spaces in UI designs to accommodate natural variations in image dimensions.

AV Work Images

AV Work images typically conform to a 3:4 (width:height) aspect ratio.

3:4 (± 10%)	Narrower	Wider	Narrowest	Widest
98%	1%	1%	1:2	9:5

Video Product Images

Video Product images typically conform to a 3:4 (width:height) aspect ratio

3:4 ($\pm 10\%$)	Narrower	Wider	Narrowest	Widest
90%	5%	5%	1:3	5:1

Contributor Images

Video Contributor images typically conform to a 6:9 (width:height) aspect ratio. Two ranges are provided due to larger variation in image dimensions.

6:9 ($\pm 20\%$)	Narrower	Wider	Narrowest	Widest
90%	0%	10%	1:3	9:5

6:9 ($\pm 10\%$)	Narrower	Wider	Narrowest	Widest
69%	1%	30%	1:3	9:5

3.11.1.5 TV Channel Logo Sizes

Possible sizes for TV channel logos are:

Width	Height
110	50
220	100

3.11.1.6 TV Program and Category Image Sizes

Possible sizes for TV Program and TV Category images are:

SIZE="NAME"	Width	Height
THUMBNAIL	75	75
SMALL	170	170
MEDIUM (default)	450	450
LARGE	720	720
XLARGE	1080	1080

3.11.2 Collaborative Artists Best Practices

The following topic provides best practices for handling collaborations in your application.

3.11.2.1 Handling Collaborations when Processing a Collection

When looking up a track using a text-based lookup, such as when initially processing a user's collection, use the following best practices:

- If the input string matches a single artist in the database, such as "Santana," associate the track in the application database with the single artist.
- If the input string matches a collaboration in the database, such as "Santana featuring Rob Thomas," associate the track in the application database with the primary collaborator and the collaboration. In this case, the Contributor, "Santana featuring Rob Thomas," will have a Contributor child, "Santana," and the track should be associated with "Santana" and "Santana featuring Rob Thomas."
- If the input string is a collaboration, but does not match a collaboration in the database, GNSDK attempts to match on the primary collaborator in the input, which would be "Santana" in this example. If the primary collaborator matches an artist in the database, the result will be the single artist. There will be an indication in the result that only part of the collaboration was matched. Associate the track in the application database with the single artist and with the original input string.

3.11.2.2 Displaying Collaborations during Playback

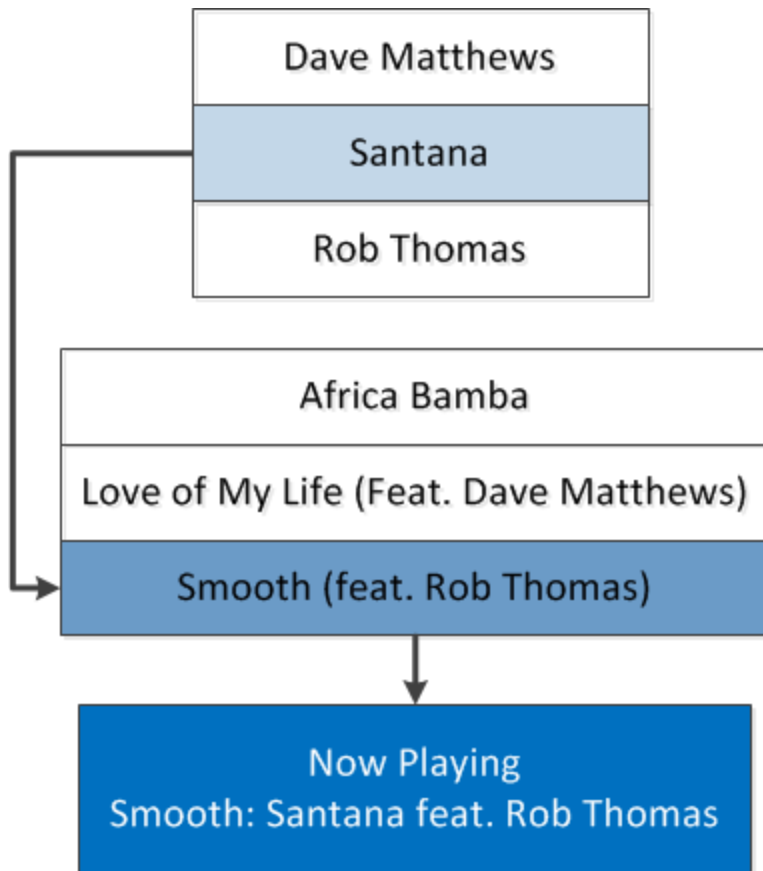
When determining what should be displayed during playback of music, use the following best practices:

- When a track by a single artist is playing, your application should display the Gracenote normalized text string. For example, when a track by Santana is playing, “Santana” should be displayed.
- When a track by a collaboration is playing, and GNSDK has matched the collaboration, the application should display the collaboration name. For example, when a track by “Santana featuring Rob Thomas” is playing, the collaboration name “Santana featuring Rob Thomas” should be displayed.
- When a track by a collaboration is playing, but only part of the collaboration was matched, Gracenote recommends that you display the original tag data for that track during playback. For example, when a track by “Santana featuring Unknown Artist” is playing, but only “Santana” was matched, the collaboration name “Santana featuring Unknown Artist” should be displayed. Gracenote recommends that you do not overwrite the original tag data.

3.11.2.3 Displaying Collaborations in Navigation

When creating navigation in your application, use the following best practices:

- If the user is navigating through the interface, and comes to “Santana” in a drop-down list, all tracks by “Santana” should be displayed, including tracks on which Santana is the primary collaborator. The list should be created using the associations that you created during the initial text-lookup phase. If the user selects “Play songs by Santana,” all songs by Santana and songs on which Santana is the primary collaborator can be played.
- Gracenote does not recommend that collaborations appear in drop-down lists of artists. For example, don’t list “Santana” and “Santana featuring Rob Thomas” in the same drop-down list. Instead, include “Santana” in the drop-down list.



3.11.2.4 Handling Collaborations in Playlists

When creating a playlist, if the user is able to select a collaboration as a seed, then only songs by that collaboration should be played. For example, if the user selects “Santana featuring Rob Thomas” as a seed for a playlist, they should only hear songs by that specific collaboration. This only applies to playlists of the form “Play songs by <artist>.” It does not apply to “More Like This” playlists, such as “Play songs like <artist>,” which use Gracenote descriptors to find similar artists.

3.11.3 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 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.

3.11.3.1 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.

3.11.3.2 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.

3.11.3.3 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.

3.11.3.4 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.

3.11.3.5 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.

3.11.3.6 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.

3.11.3.7 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.

3.12 Deploying Android Applications

To access GNSDK in an Android application, add the GNSDK jar and native shared libraries to your application's libs folder. Copy the jar libraries directly into the libs folder. Copy the native shared libraries into the architecture sub-folders under libs.

The table below shows the GNSDK Android libraries and their location in the package

Jar	Location
gnsdk.jar and gnsdk_helpers.jar	.../wrappers/gnsdk_java/jar/android
libgnsdk_marshall.so	.../wrappers/gnsdk_java/lib/android_*
libgnsdk_*.so	.../lib/android_*

GNSDK uses GABI++ for C++ support and requires that libgabi++_shared.so is included in the application's architecture sub-folder under libs. The libgabi++_shared.so is an Android library provided in the Android NDK

3.12.1 GNSDK Android Permissions

To use the GNSDK properly in your Android application, configure the application with the follow settings:

- Record audio
- Write to external storage
- Access Internet
- Access network state

Add these permissions to the Android application's AndroidManifest.xml file.

Chapter 4 Data Models

This section describes the data elements, attributes, and values used in GNSDK applications.

4.1 Data Models

The following table links to GNSDK data models for the corresponding query response type:

- **Music**
 - GnResponseAlbums
 - GnResponseTracks
 - GnResponseContributors
- **MusicID-Match**
 - GnResponseDataMatches

Chapter 5 API Reference

This section provides reference information about GNSDK APIs. For details about how to use these APIs, see [Develop and Implement](#).



Note: For a list of third-party and open-source (OSS) licenses used by GNSDK, open the /docs/license-files folder in the software package.

5.1 API Reference Overview

The following API Reference documentation is available for GNSDK.

API	Location in Package
C++ API Reference	docs/html-docs/html-oo/Content/api_ref_cplusplus/index.html
C# API Reference	docs/html-docs/html-oo/Content/api_ref_csharp/index.html
Java (J2SE) API Reference	docs/html-docs/html-oo/Content/api_ref_java_j2se/html/index.html
Java (Android) API Reference	docs/html-docs/html-oo/Content/api_ref_java_android/html/index.html



Note: For a list of third-party and open-source (OSS) licenses used by GNSDK, see the /docs/license-files folder in the software package.

5.2 Playlist PDL Specification

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 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.

5.2.1 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.

5.2.1.1 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 idents. 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>]]
```

5.2.1.2 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	+, -, *, /

5.2.1.3 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.

5.2.1.4 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 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

5.2.1.5 Functions

PDL supports these functions:

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

5.2.1.6 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.

5.2.2 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"
```

Glossary

A

Album

A collection of audio recordings, typically of songs or instrumentals.

Artist

The person or group primarily responsible for creating the Album or Track.

Audio Work

A collection of classical music recordings.

AV Work

In general, the terms Audio-Visual Work, Work, and AV Work are interchangeable. A

Work refers to the artistic creation and production of a Film, TV Series, or other form of video content. The same Work can be released on multiple Product formats across territories. For example, The Dark Knight Work can be released on a Blu-ray Product in multiple countries. A TV Series such as Lost is also a Work. Each individual episode comprising the Series is also a unique Work. Although the majority of Works are commercially released as a Product, not all Works have a Product counterpart. For example, a TV episode which airs on TV, but is not released on DVD or Blu-ray is considered a Work to which no Product exists.

C

Chapter

A Video feature may contain chapters for easy navigation and as bookmarks for partial viewing.

Character

An imaginary person represented in a Video Work of fiction. Feature - A video feature has a full-length running time usually between 60 and 120 minutes. A feature is the main component of a DVD or Blu-ray disc which may, in addition, contain extra, or bonus, video clips and features.

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.

Contributor

A person that participated in the creation of the Album or Track.

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 film studio) to a Video Work.

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

E

Episode

A specific instance of a TV Program in a TV Series.

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).

Filmography

All of the Works associated with a Contributor in the Gracenote Service, for example: All Works that are linked to Tom Hanks.
Franchise - A collection of related Video Works. For example: Batman, Friends, Star Wars, and CSI.

Filmography

All of the Works associated with a Contributor in the Gracenote Service, for example: All Works that are linked to Tom Hanks.

Franchise

A collection of related Works. Each of the following examples is a unique franchise: Batman Friends Star Wars CSI

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.

Mediography

All of the Works associated with a Contributor in the Gracenote Service, for example: All works that are linked to Tom Hanks.

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-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.

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.

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

Product

A Product refers to the commercial release of a Film, TV Series, or video content. Products contain a unique commercial code such as a UPC, Hinban, or EAN.

Products are for the most part released on a physical format, such as a DVD or Blu-ray.

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

Season

An ordered collection of Works, typically representing a season of a TV series. For example: CSI: Miami (Season One), CSI: Miami (Season Two), CSI: Miami (Season Three)

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.

Series

A collection of related Works, typically in sequence, and

often comprised of Seasons (generally for television series); for example, CSI Las Vegas. A Series object may have varying structures of Episodes and Seasons objects. Three common Series object structure hierarchies are - Series object contains only Episode objects; for example, Ken Burns' The Civil War series. Series object contains multiple Seasons objects that contain multiple Episodes objects; for example, the animated television series The Simpsons. Series object contains one or more independent Episodes objects (meaning, not contained within Seasons objects) and multiple Seasons objects that contain multiple Episodes objects. This structure occurs for cases when a pilot episode (represented by an independent Episode object) is developed into a series. An example of this is Cartoon Network's Samurai Jack series, which was initially launched as a television movie and then later developed into a television series.

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.

Title

Also referred to as Title Set. DVDs and Blu-ray discs may contain multiple titles. A typical movie DVD may be comprised of multiple titles, one of which comprises the main feature (in this case, the movie) and is referred to as the main title. Other titles, or extras, are often comprised of previews, behind-the-scenes documentaries, or other content.

TOC

Table of Contents. An area on CDs, DVDs, and Blu-ray discs that describes the unique track layout of the disc.

TOP

Table of Programs. The list of program addresses found on a DVD that allows the GN

MusicID system to find the DVD in the MDB.

Track

A song or instrumental recording.

V

Video Clip

A short video presentation.

Video Disc

A video disc can be either DVD (Digital Video Disc) or Blu-ray. DVD is an optical disc storage format, invented and developed by Philips, Sony, Toshiba, and Panasonic in 1995. DVDs offer higher storage capacity than Compact Discs while having the same dimensions. Blu-ray is an optical disc format designed to display high definition video and store large amounts of data. The name Blu-ray Disc refers to the blue laser used to read the disc, which allows information to be stored at a greater density than is possible with the longer-wavelength red laser used for DVDs.

Video Explore

Provides extended video recognition and searching, enabling user exploration and discovery features.

Video ID Module

Provides video item recognition for DVD and Blu-ray products via TOCs.

Video Layer

Both DVDs and Blu-ray Discs can be dual layer. These discs are only writable on one side of the disc, but contain two layers on that single side for writing data to. Dual-Layer recordable DVDs come in two formats: DVD-R DL and DVD+R DL. They can hold up to 8.5GB on the two layers. Dual-layer Blu-ray discs can store 50 GB of data (25GB on each layer).

Video Side

Both DVDs and Blu-ray discs can be dual side. Double-Sided discs include a single layer on each side of the disc that data can be recorded to. Double-Sided recordable

DVDs come in two formats: DVD-R and DVD+R, including the rewritable DVD-RW and DVD+RW. These discs can hold about 8.75GB of data if you burn to both sides. Dual-side Blu-ray discs can store 50 GB of data (25GB on each side).

W

Work

See AV Work

Index

A

ACR [58, 67](#)

Album [19, 23, 26, 32, 36, 56, 177, 194, 201, 228](#)

AlbumID [28, 31, 124](#)

Android [63, 72, 74, 83, 99, 107, 125, 147, 164, 220, 224](#)

API Reference [160, 223-224](#)

Artist [15, 19, 32, 51, 54, 97, 120, 154, 201, 213, 228](#)

 Image [168](#)

 Name [21, 97, 120, 201, 228](#)

Audio [24, 104, 123, 217](#)

AV Work [15, 55, 213](#)

B

Batch [121](#)

Behavior [79](#)

Best Practices [26, 29, 55, 80, 120, 189, 209, 215, 217](#)

Bundles [99, 148](#)

C

C# API [224](#)

C++ API [224](#)

Callback [83](#)

Catalog [20](#)

CD TOC [24, 30, 59, 89, 104-105, 122](#)

Channel [214](#)

Classical Music [19, 170](#)

Client ID [61-62](#)

Client Tag [66](#)

Collaborative Artists [215](#)

Collection [33, 35, 37, 175, 191, 215](#)

Contributor [55](#), [120](#), [213](#), [215](#)

Cover Art [53](#), [56](#), [168](#), [212](#)

D

Data Model [160](#), [170](#)

Database [98](#), [110](#)

Descriptor [73](#)

Dimensions [52](#), [195](#), [209](#)

DSP [27](#), [59-60](#), [192](#), [204](#)

E

Enriched Content [161](#)

Enriched Metadata [18](#)

Enrichment [24](#), [32](#)

Enumerate [191](#)

Environment [84](#)

EPG [58](#), [73](#)

Equivalency [35](#)

Era [19-20](#), [35](#), [173](#), [184](#), [201](#), [228](#)

Event [83](#)

Events [208](#)

F

Fetch [161](#)

Find [194](#)

Fingerprint [26](#), [98](#), [104](#)

G

GDB [77](#)

GDO [30](#), [36](#), [122](#), [168](#), [177](#), [200](#),
[228](#)

Genre [15](#), [23](#), [35](#), [44](#), [51](#), [54](#), [166](#),
[171](#), [184](#), [201](#), [228](#)

GnDataObject [168](#), [200](#), [228](#)

GnMic [104](#), [147](#)

GOET [201](#), [228](#)

Gracenote [ii](#), [14-15](#), [18-19](#), [22-24](#),
[26](#), [30](#), [32-33](#), [36](#), [38](#), [41](#), [50](#),
[52](#), [55](#), [58](#), [61-62](#), [72](#), [83](#), [86](#),
[94](#), [98](#), [104-105](#), [121](#), [129](#),
[147](#), [158](#), [161](#), [171](#), [175](#), [191](#),
[199](#), [204](#), [209](#), [216-217](#), [227](#)

Gracenote Media Elements [15](#)

Groups [18](#), [77](#)

H

Header Files [62](#)

Hierarchical Groups [18](#)

I

Identifiers [22](#), [36](#), [194](#)

Identify [31](#)

Image

Dimensions [53](#), [209](#)

Formats [52](#), [211](#)

Images [53](#), [212](#)

Initialize [61](#), [65](#), [101](#), [111](#), [137](#), [179](#)

Instantiate [61-62](#), [84](#), [107](#), [124](#),
[147](#), [179](#), [204](#)

iOS [67](#), [147](#)

Iterate [187](#), [194](#), [206](#)

L

LANG [173](#)

Language [33](#), [59](#), [73](#), [176](#), [197](#),
[224](#)

LibraryID [28](#), [124](#)

License File [63](#)

Link [22](#), [32](#), [58](#), [168](#)

List [15](#), [21](#), [127](#)

Literals [200](#), [227](#)

Local [58](#), [69](#), [86](#), [94](#), [98](#), [107](#), [148](#),
[179](#)

Local Lookup [107](#)

Local Storage [179](#)

Locale [61](#), [72](#), [158](#), [195](#)

Locale-Dependent [75](#), [161](#)

Log [100](#), [134](#), [150](#)

M

Manager [58](#), [61](#), [161](#), [192](#), [200](#),
[204](#), [228](#)

Match [26](#), [52](#), [120](#), [221](#)

Media [15](#), [23](#), [36](#), [53](#), [57](#), [62](#), [110](#),
[133](#), [211](#)

Memory [92](#)

Metadata [18-19](#), [27](#), [36](#), [52](#), [125](#),
[158](#), [170](#)

Mood [19](#), [23](#), [35-36](#), [38](#), [40](#), [44](#),
[184](#), [191](#), [201](#), [228](#)

MoodGrid [59](#)

Multi-Threaded Access [78](#)

Multiple TOC Matches [25](#)

Multiple TOCs and Fuzzy
Matching [25](#)

Music [14-15](#), [19](#), [22](#), [26](#), [32](#), [53](#), [59](#),
[72](#), [104-105](#), [121](#), [146](#), [161](#),
[170](#), [212](#), [221](#)

MusicID [23-24](#), [26-27](#), [30-31](#), [34](#),
[36](#), [38](#), [51-52](#), [59](#), [62](#), [98](#), [105](#),
[120](#), [123](#), [146](#), [177](#), [192](#), [204](#),
[217](#), [221](#)

MusicID-File [27](#), [30](#), [34](#), [36](#), [52](#), [59](#),
[105](#), [123](#), [177](#)

N

Navigating [40](#)

Needs Decision [158](#)

Number [35](#), [171](#)

O

Order [198](#), [225](#)

Origin [184](#), [201](#), [228](#)

Overview [23](#), [27](#), [38](#), [50](#), [52](#), [58](#),
[124](#), [191](#), [224](#)

P

Parse [62](#)

Partial [159](#)

PCM [111](#), [124](#), [147](#)

PDL [37](#), [176](#), [197](#), [224](#)

Performance [189](#)

Permissions [220](#)

Physical Media [190](#)

Play [50](#)

Playlist [19](#), [33](#), [35](#), [43](#), [51](#), [59](#), [62](#),
[73](#), [175](#), [191](#), [197](#), [204](#), [224](#)

Playlist Description Language
(PDL) [176](#)

Populate [175](#)

Product [15](#), [25](#), [55](#), [133](#), [213](#)

Program [214](#)

Provider [79](#), [87](#), [103](#)

Providers [191](#)

Python [119](#)

Q

Query [52](#), [88](#), [96](#), [121](#), [149](#), [161](#),
[183](#), [195](#), [197](#), [204](#), [224](#)

Queue [208](#)

R

Radio [50](#), [148](#), [173](#)

RAM [79](#)

Region [73](#)

Request [205](#)

Requirements [33](#), [209](#)

Response [28](#)

Results [44](#), [87](#), [125](#), [147](#), [158](#), [184](#),
[197](#)

Rhythm [48](#), [50](#), [60](#), [203](#)

S

Save [169](#), [209](#)

Score [202](#), [230](#)

Search [197](#), [224](#)

Season [15](#)

Seed [37](#), [199](#), [226](#)

Series [15](#)

Size [163](#)

Sonic Attributes [19](#)

Sources [192](#)

SQLite [34](#), [60](#), [78](#), [87](#), [98](#), [107](#), [179](#),
[192](#)

Station [50](#)

Status [73](#), [125](#)

Storage [34](#), [37](#), [67](#), [86](#), [179](#)

Storage Provider [79](#), [103](#)

Streaming Audio [104](#), [149](#)

Submit [60](#)

T

Tempo [19](#), [35-36](#), [184](#), [201](#), [228](#)

Text [24](#), [26](#), [59](#), [105](#), [120-121](#), [127](#),
[153](#), [201](#), [228](#)

Third-Party

Identifiers [22](#)

Title [19-20](#), [51](#), [97](#), [120](#), [127](#), [154](#),
[166](#), [171](#)

TOC [104](#)

Track [19](#), [26](#), [35-36](#), [51](#), [97](#), [108](#),
[120](#), [122](#), [154](#), [177](#)

TrackID [28](#), [124](#)

Tuning [208](#)

TV Channel [214](#)

TV Program [214](#)

U

Update [78](#)

User [50](#), [61](#), [124](#), [147](#), [159](#), [204](#),
[219](#)

Object [61-62](#)

V

Values [15](#), [40](#), [44](#)

Video [15](#), [54](#), [73](#), [212](#)

Product [15](#), [55](#), [213](#)

Voice Commands [44](#)

W

Windows *64, 74, 83, 88, 103, 108, 127, 153, 166*

Windows Phone *64, 74, 88, 103, 108, 127, 153, 166*

Work *15, 20, 55, 213*