gracenote.

Mobile Client Android Implementation Guide

Product Version 3.2.0.144

Published: 23-Jul-2013 09:56

Table of Contents

Overview

This document is the Implementation Guide to the application programming interface (API) of the Gracenote Mobile Client software library. It provides conceptual and background information, implementation guidance, and example code to aid software developers in building application programs incorporating Gracenote Mobile Client services. For complete reference information on the Mobile Client API, see the included Javadoc.

> **ⓘ Note**
>
> For conceptual simplicity, error checking has been omitted from most of the programming examples in this manual. In an actual production application, you will of course want to check the returned result code after each library call and terminate the logic flow or take appropriate recovery measures in case of failure. For more complete example code including full error checking, see the Sample Application included with the Mobile Client distribution package.

Deployment

Mobile Client is delivered as an Eclipse project that can be integrated into an Android development environment that uses the Eclipse integrated development environment (IDE).

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

Integrating Mobile Client into an Existing Android Project

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

- `GN_Music_SDK.jar` (Java library)
- `libgnmc_decoder.<version number>.so` (native library)
- `libgnmc_fpx.<version number>.so` (native library)
- `libgnmc_id3tag.<version number>.so` (native library)
- `libgnmc_aactag.<version number>.so` (native library)
- `libgnencryption.<version number>.so`
- `libgnlocal_lookup.<version number>.so`
- `libgntransition_detector.<version number>.so`

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

Mobile Client Android Permissions

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

- Record audio
- Access fine (e.g., GPS) location (only required for Sample app and if your app is using GPS feature)
- Write to external storage
- Access Internet
- Access network

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

Migrating to this Release

Supported Sampling Rates for MusicID-Stream and MusicID-File

As of Release 3.1, the supported sampling rates for MusicID-Stream and MusicID-File are:

- 8000 Hz
- 44100 Hz

Library Update

The following libraries must be copied from the Mobile Client distribution package into your application:

- `GN_Music_SDK.jar` (Java library)
- `libgnmc_decoder.<version number>.so` (native library)
- `libgnmc_fpx.<version number>.so` (native library)
- `libgnmc_id3tag.<version number>.so` (native library)
- `libgnmc_aactag.<version number>.so` (native library)
- `libgnencryption.<version number>.so`
- `libgnlocal_lookup.<version number>.so`
- `libgntransition_detector.<version number>.so`

> ⚠ Versioning (<version-number>) is broken down as follows:
>
> <library>.<M.N.I.B>.so
>
> where
>
> M - Major release number
>
> N - Minor release number
>
> I - Improvement release number
>
> B - Build number
>
> For example: `libgnmc_decoder.2.5.9.1.so`
>
> Note that **backward compatibility between different library versions is NOT supported.**

The libraries were renamed in Mobile Client 2.5.6. The table below summarizes the changes made:

Old Name	New Name
`libdec.so`	`libgnmc_decoder.<version number>.so`
`libstream_core.so`	`libgnmc_fpx.<version number>.so`
`libid3tag.so`	`libgnmc_id3tag.<version number>.so`
`libaactag.so`	`libgnmc_aactag.<version number>.so`

GNConfig Parameter Changes

Several `GNConfig` parameters were renamed in Mobile Client 2.5.2 to improve consistency and extensibility. As of 3.1, you **must** migrate your code to match the new naming conventions.

The tables below summarize the changes made, including the parameters that are no longer supported.

No Longer Supported

No Longer Supported Name	New Name	Comments
country	content.country	As of Version 2.5.8., the default for this is `null`, before, it was `USA`
genre and genreId properties in GNSearchResponse	trackGenre and albumGenre	As of Version 2.5: Replace `GNSearchResponse` properties genre with `trackGenre` and `genreId` with `albumGenre`. Multiple genre levels can now be returned, so genres are delivered as a collection of GNDescriptor objects that contain a genre descriptor and a genre identifier. The properties `genre` and `genreId` will **NO LONGER** return the descriptor and identifier from the lowest level genre returned to Mobile Client.

isGenreCoverArtEnabled	content.coverArt.genreCoverArt	
lang	content.lang	
preferredLinkSource	content.link.preferredSource	
webservices.coverArtSizePreference	content.coverArt.sizePreference	
webservices.gzipEnabled	N/A	
webservices.isInlineCoverArtEnabled	content.coverArt	As of Version 2.5. The default for this is `false` (as of version 2.5.8)
webservices.isSingleBestMatchPreferred	content.musicId.queryPreference.singleBestMatch	

Case Change Only

Cases are changed for consistency. Older case is supported. No deprecation.

Old Name	New Name
Content.contributor.images	content.contributor.images
Content.review	content.review
Content.contributor.biography	content.contributor.biography

Changed Default Values

Name	New Default Value	As of
content.coverArt	`false`	Version 2.5.8
content.country	`null`	Version 2.5.8 Prior to this, the default value was `USA`

New GNConfig Property

Property	Type	Default	As of	Description
content.allowfullresponse	Boolean string	False	3.1	Indicates if Gracenote response from local cache identification should contain full or partial metadata. The following operations use local cache lookup: 1. `GNRecognizeStream.idNow` 2. `GNOperations.recognizeMIDStreamFromMic` 3. `GNOperations.recognizeMIDStreamFromPcm`

GNStatus Change

The `GNStatus` RECORDING value has been removed and replaced with LISTENING.

Deprecated GNSearchResponse

Deprecated GNSearchResponse Methods

Old Name	New Name
getArtistImage()	getContributorImage()

New GNSearchResponse Property

Property	Type	As of	Description
partial	Boolean	3.1	Indicates if response contains full or partial metadata.

Technical Requirements

Hardware	Android ARMv6- or ARMv7-capable/compatible
Platform	Android 2.2 or higher
Java Development Kit	J2SE 5.0 (JDK 1.5) or higher
Code size (JAR + native libraries)	< 7.5 MB

Configuration and Authentication

Mobile Client uses a *configuration object* of type GNConfig to control its behavior. The behavior can be modified by altering the configuration object.

To obtain a configuration object, use the GNConfig.init method. This method returns a GNConfig instance that must be stored and used with Gracenote operations (see the section). The method accepts a *client identifier* provided by your Gracenote Professional Services representative:

```
Context applicationContext;
GNConfig config;

// Obtain context from Android Activity object
applicationContext = this.getApplicationContext();

// Generate configuration object
config = GNConfig.init("12345678-ABCDEFGHIJKLMNOPQRSTUVWXYZ012345", applicationContext);
```

The application context can be obtained from the Android Activity object when its onCreate method is called. The client identifier is used to generate authentication information, which is stored in the object and used in turn to gain access to Gracenote's cloud-based services.

You can customize a configuration object and set its properties with the `GNConfig.setProperties` method. For example, you can configure the preferred language for metadata returned from Gracenote:

```
// Set preferred language to Japanese
config.setProperty ("content.lang","jpn");
```

See the *Android Mobile Client API Reference Guide > Class GNConfig* for a complete list of customizable properties.

Operations

An *operation* is a request performed by Mobile Client, such as creating a fingerprint or recognizing an audio stream. The Mobile Client classes `GNOperations` and GNRecognizeStream provide methods for invoking operations.

Invoking Operations

GNOperations and GNRecognizeStream are based on Android AsyncTask*. Operations run asynchronously to the application invoking them, and return their results via a mechanism known as a *result-ready object* (described below under). Each operation must be provided a configuration object generated by `GNConfig.init`, along with a result-ready object to receive the results:

```
// Create result-ready object to receive recognition result
ApplicationSearchResultReady searchResultReady = new ApplicationSearchResultReady();

// Invoke recognition operation
GNOperations.recognizeMIDStreamFromMic (searchResultReady, config);
```

> **ⓘ Note**
>
> Operations call result-ready and status-changed methods in the application's main thread. As a best practice, any time-consuming or complex computational processes should not be run in the main thread. Doing so will block the UI and may cause the application to behave poorly and impact performance. For example, retrieving cover art should be run in a background thread.
>
> *GNOperations and GNRecognizeStream methods should not be invoked inside another AsyncTask's doInBackground method.

Receiving Results

Result-ready objects implement one of the following Mobile Client interfaces, depending on the type of operation:

- `GNFingerprintResultReady`
- `GNSearchResultReady`

Mobile Client calls the result-ready object's `GNResultReady` method when a result is generated. Your application can use this method to process the result:

```
// Result-ready object implements GNSearchResultReady interface
class TrackSearchTask implements GNSearchResultReady
{
    // Provide implementation for GNResultReady method
    public void GNResultReady (GNSearchResult result)
    {
      // Application code to process operation result
    }


  public void trackSearch(String trackId)
  {
   GNOperations.fetchByTrackId(this, config, trackId);
  }

  private GNConfig config;
}
```

Audio Recognition

Mobile Client provides three Gracenote technologies for audio recognition:

1. **MusicID-Stream**, for audio delivered from a microphone or other streaming audio source (such as a radio signal or streaming Internet source).
2. **MusicID-File**, for audio extracted from an audio file (such as a `.wav` or `.mp3` file).
3. **AlbumID**, for identifying groups of audio files using fingerprints, text inputs, tag data, and Gracenote Identifiers. AlbumID can also use this data to group files into albums.

The results of recognition operations are returned to the application via a result-ready object implementing the `GNSearchResultReady` interface.

> ℹ️ **Note**
>
> If requested (and if the application is so entitled), Mobile Client can also provide cover art and Link identifiers with the recognition results; see Cover Art (see page 37) and Retrieving Link Data (see page 46), below, for further information.

> ✅ Stream-based recognition can take longer than file-based. When recognizing an audio file, use file-based recognition to obtain the best response time.

Setting Up a Cache of Music Metadata for MusicID Stream

Gracenote provides a local database of fingerprints and metadata for some example songs. The SDK uses this database for local identification. If that fails, then an online lookup is attempted.

Beginning with Mobile Client 3.1, Gracenote makes available a *bundle* of fingerprints and metadata for currently trending tracks that your application can download. Contact Gracenote Professional Services for the download site. Once downloaded, your application can *ingest* the bundle and create a cache with methods provided through the `GnConfig` class - `loadCache()` and `clearCache()`.

> ⚠ The `clearCache()` API has two signatures:
>
>   1. `GNCacheStatusEnum clearCache()`
>   2. `GNCacheStatusEnum clearCache(boolean vacuumDB)`
>
> As its parameter implies, the second call gives your application the option to *vacuum* (de-fragment and compact) the local database. On a `clearCache()` call, your application should only invoke a vacuum if you are NOT going to do a subsequent `loadCache()` call and the storage space needs to be reclaimed. Calling `clearCache` without vacuuming and then `loadCache(new_bundle_file_path)` will not increase the current database size unless the new bundle requires additional space. Vacuuming the database can take 5 or more seconds and it is recommended your application do this on a background thread. For a local database containing 1000 tracks, vacuuming can save approximately 22MB in device memory.

Note that matching from cache can significantly improve SDK performance compared to online lookups. In most quad-core and dual-core devices, recognition time from local cache is considerably faster.

Loading cache example:

```
//
// Example - from sample application GracenoteMusicID.java
//
    private void setUpCache() {
        new Thread(new Runnable() {
            public void run() {
                try {
                    InputStream in = getResources().openRawResource(R.raw.bundle161);
                    final String bundleFilePath =
Environment.getExternalStorageDirectory().getAbsolutePath() + "/bundle161.b";
                    FileOutputStream out = new FileOutputStream(bundleFilePath);
                    byte[] buff = new byte[1024];
                    int read = 0;
                    try {
                        while ((read = in.read(buff)) > 0) {
                            out.write(buff, 0, read);
                        }
                    } finally {
                        in.close();
                        out.close();
                    }
                    GNCacheStatusEnum cacheStatus = config.loadCache(bundleFilePath);
                    if(cacheStatus != GNCacheStatusEnum.SUCCESS){
                        Log.e(LOG_TAG, "load cache failed with status: " + cacheStatus.toString());
                    }
                    // Delete Bundle after loading cache
                    File bundlefile = new File(bundleFilePath);
                    if (bundlefile.exists()) {
                        bundlefile.delete();
                    }
                } catch (Exception e) {
                    Log.e(LOG_TAG, e.toString());
                }
            }
        }).start();
    }
```

To implement recognition from local audio fingerprint storage:

1. Contact Gracenote Professional Services for the bundle download site.
2. In your application, download the bundle to device storage.
3. In your application, call the `GNConfig.loadCache()` method to set up a cache from the bundle. As a best practice, call this method in a background thread.
4. Delete the bundle from device storage when done loading cache.

⚠ On slower devices (e.g., those around 600MHz, single-core CPU), `loadCache()` should be a blocking operation for large (1000 or greater track) bundles.

Full and Partial Metadata Responses From Cache

In a local lookup (cache-based identification) request, you can indicate if your app is returned a full or partial metadata response with the `GNConfig.allowfullresponse` field (new in 3.1). In a partial metadata response, the following fields are returned:

- Album title
- Artist (track level)
- Track title
- Track duration
- Track match position
- Track GNID (Gracenote ID)
- Album GNID

The `GNSearchResponse. partial` flag (also new in 3.1) indicates if your response contains full or partial metadata results. Note that this flag will only be true if a single, best match response is returned.

To get full metadata after a partial response, your app would need to call `GNOperations.fetchByTrackId` or `fetchByAlbumId`.

> ⓘ Since the local cache only contains partial metadata, a locally-identified match requires an additional online lookup for full metadata. Doing this requires Internet access and an additional 800ms -1 second to resolve the request. If `GNConfig.allowfullresponse` is `true`, and you do NOT have an Internet connection, your app will receive an "Internet Connection" error.
>
> Note that it is possible for `partial` and `allowFullResponse` to both be `true`. This can occur if the network is not available and metadata is returned only from cache. Your application should check the value of `partial` EVEN IF `allowFullResponse` is `true` and not assume full metadata.

The following operations use local cache lookup:

1. GNRecognizeStream.idNow
2. GNOperations.recognizeMIDStreamFromMic
3. GNOperations.recognizeMIDStreamFromPcm

MusicID-Stream

MusicID-Stream can be used to recognize a snippet of a song, such as a recording received from the device microphone or from an Internet stream.

Mobile Client provides two methods for invoking a MusicID-Steam recognition, one designed for simplicity, the other for flexibility.

Supported Sampling Rates for MusicID-Stream:

- 8000 Hz
- 44100 Hz

GNAudioSourceMic

This class provides simple microphone management, optimized for `GNRecognizeStream` and has two methods:

- `startRecording()` – start or resume recording
- `stopRecording()` - stop recording and release microphone

The `GNAudioSourceMic` constructor takes a `GNAudioConfig` instance and an object that implements the `GNAudioSourceDelegate` interface:

```
// From the Sammple Application
public class GracenoteMusicID extends Activity implements GNSearchResultReady,GNAudioSourceDelegate {

// ...
GNAudioConfig myDeviceAudioConfig;
GNAudioSourceMic mAudioSource;
// ...
// Initialize with recommended audio settings
int sampleRate = 44100;
int bytesPerSample = 2;
int numChannels = 1;

this.myDeviceAudioConfig = new GNAudioConfig(sampleRate, bytesPerSample,numChannels);
mAudioSource = new GNAudioSourceMic(this.myDeviceAudioConfig, this);
try {
    mAudioSource.startRecording();
}
catch (Exception e) {
 //audio-recording initialization failed
}
// ...
```

The `GNAudioSourceDelegate` interface requires an `audioBufferReady()` method in order to receive recorded audio samples from `GNAudioSourceMic` as it becomes available. This method can be used to pass audio samples to `GNRecognizeStream` as shown below:

```
GNRecognizeStream gnRecognizeStream;

// ...

public void audioBufferReady(byte[] samples, boolean audioInputError) {
?
 if(!audioInputError){
   gnRecognizeStream.writeBytes(samples);
     } else {
      //error occurred while reading input from microphone
     }
}
```

NOTE: It is recommended that your application use this class when streaming audio from the device microphone.

GNRecognizeStream

The `GNRecognizeStream` class provides APIs to quickly recognize audio streamed from a device microphone or other input source. Using this class, a Mobile Client application continuously processes PCM data from the input stream and prepares it for a recognition operation.

The sample application calls the `idNow()` method to initiate the audio recognition process. All results are delivered as a result-ready object implementing the `GNSearchResultReady` interface. During the audio recognition process, Mobile Client sends operation progress status updates to the application.

The following steps illustrate how to use the `GNRecognizeStream class when streaming from the device microphone:`

1. Initialize `GNConfig`. If supporting local lookups, set up the local fingerprint database by calling load `Cache`. For more information,
2. Initialize a `GNAudioConfig` object with these recommended settings:
   - Number of channels: mono (1) or stereo (2)
   - Sample rate: 44.1K (44100.0)
   - Bytes per sample: 16-bit mono(2) or 16-bit stereo (4)
3. Initialize `GNAudioSourceMic` and implement `GNAudioSourceDelegate`
4. Implement `GNSearchResult` to receive recognition results. The Mobile Client SDK invokes this in the main thread when a search result is ready.
5. Call `GNRecognizeStream.startSession(),` passing in your `GNSearchResultReady` object and `GNAudioConfig` object.
6. Call `GNAudioSourceMic.startRecording()`
7. As audio input is buffered continuously, feed in the samples for recognition using the `GNRecognizeStream.writeBytes` method.
8. Implement a recognition operation (`idNow`) when user taps an **ID Now** button.
9. Once a match is found, your app receives a `GNSearchResult` object in its `GNSearchResultReady` callback.
10. Call `GNAudioSourceMic.stopRecording` and `GNRecognizeStream.stopSession` to stop the recording and streaming session when your application goes into the background. When the application resumes, call `GNRecognizeStream.startSession` and `GNAudioSourceMic.startRecording` again.

> **ⓘ Note**
> After `idNow` is called, the application remains in "request mode" (listening, fingerprinting, or recognizing) until the operation is complete. During this time, repeated requests (such as repeated tapping of an **ID Now** button) are ignored. Your application must first call the `cancelIdNow` method to abandon the operation in progress before a new idNow operation can be initiated.

Best Practices for GNRecognizeStream

Follow these guidelines when implementing MusicID-Stream recognition with `GNRecognizeStream`.

- When streaming from the device microphone, use `GNAudioSourceMic` instead of the Android platform's `AudioRecord` class.
- Gracenote recommends using an audio buffer size of 2048 bytes for best matches.
- Once started, the recognition session will continuously buffer audio received from the microphone until you stop the session. To handle session interruptions (e.g. when the application activity is paused as the user receives a phone call, or when the application is pushed into the background), the activity's onPause, onStop, and onResume methods should include operations indicated in step 10 above.

GNOperations.recognizeMIDStreamFromMic

`GNOperations.recognizeMIDStreamFromMic` recognizes audio recorded from the microphone and is the simplest way to recognize music playing in the user's environment.

> ⚠ If the application can take advantage of pre-buffering audio prior to an identification request, Gracenote recommends you use `GNRecognizeStream.idNow` instead of `GNOperations.recognizeMIDStreamFromMic`. The `idNow` operation can return results much faster since GNRecognizeStream supports continuous audio pre-buffering.

When invoked, Mobile Client obtains the device microphone and records 6.5 seconds of audio. This audio is processed and a MusicID-Stream fingerprint is generated. The fingerprint is then submitted to Gracenote Web Services for recognition. The result is delivered via an object that implements the `GNSearchResultReady` interface.
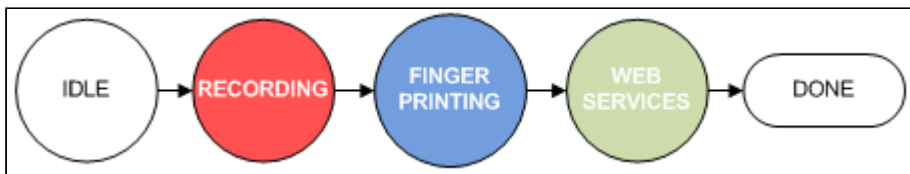
Status flow in stream-based audio recognition

The following example shows how to invoke `GNOperations.recognizeMIDStreamFromMic`.

```
// Create result-ready object to receive recognition result.
// ApplicationSearchResultReady must implement the GNSearchResultReady interface
ApplicationSearchResultReady searchResultReady = new ApplicationSearchResultReady();

// Invoke recognition operation with the result-ready object and a GNConfig object
// instance
GNOperations.recognizeMIDStreamFromMic (searchResultReady, config);
```

During audio recognition, Mobile Client sends status updates to notify the application of progress.

> ℹ **Note**
>
> - No more than one application can access the microphone at a time; consequently, recording is a blocking function. The Android platform blocks attempts to access a microphone that is already in use (except in the case of incoming phone calls).
> - If recognition from microphone recording is a frequent operation, consider using GNRecognizeStream instead of GNOperations.recognizeMIDStreamFromMic.

GNOperations.recognizeMIDStreamFromPcm

`GNOperations.recognizeMIDStreamFromPcm` recognizes audio provided as a buffer of PCM (pulse-code modulation) data. This provides the application with additional flexibility: for instance, the application can recognize audio from external streaming audio sources such as a radio broadcast or an Internet stream.

When invoked, Mobile Client reads the PCM audio data. The audio is processed and a MusicID-Stream fingerprint is generated. The fingerprint is then submitted to Gracenote Web Services for recognition. The result is delivered via an object that implements the `GNSearchResultReady` interface.
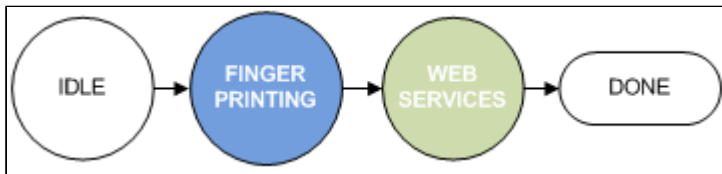
Status flow in PCM-based audio recognition

The following example shows how to invoke `GNOperations.recognizeMIDStreamFromPcm`.

```
// Create PCM sample buffer
GNSampleBuffer sampleBuffer = new GNSampleBuffer(samples, bytesPerSample, numChannels, sampleRate);

// Create result-ready object to receive recognition result
// ApplicationSearchResultReady must implement the GNSearchResultReady interface
ApplicationSearchResultReady searchResultReady = new ApplicationSearchResultReady();

// Invoke recognition operation with the result-ready object, a GNConfig object
// instance and the PCM buffer
GNOperations.recognizeMIDFileFromPcm (searchResultReady, config, sampleBuffer);
```

During audio recognition, Mobile Client sends status updates to notify the application of progress.

> ℹ **Note**
>
>   The PCM audio sample must be at least 6.5 seconds long for Mobile Client to sucessfully generate a MusicID-Stream fingerprint.

MusicID-File

MusicID-File can be used to recognize an audio file.

Mobile Client provides two methods for invoking a MusicID-File recognition, one designed for simplicity, the other for flexibility.

GNOperations.recognizeMIDFileFromFile

`GNOperations.recognizeMIDFileFromFile` recognizes and audio file stored on the device and is the simplest way to perform a MusicID-File recognition.

When invoked Mobile Client decodes the audio file. The audio is processed and a MusicID-File fingerprint is generated. The fingerprint is then submitted to Gracenote Web Services for recognition. The result is delivered via an object that implements the `GNSearchResultReady` interface.

Status flow in file-based audio recognition

Mobile Client can recognize audio files in the following formats:

- `.wav`
- `.mp3`
- `.aac`

> ⚠ MPEG-2 AAC is **NOT** supported, since Android does not support MPEG-2 AAC. See
> http://developer.android.com/guide/appendix/media-formats.html for more information.

The following sampling rates are supported, in both monaural and stereo:

- 8000 Hz
- 44100 Hz

> ⚠ Files containing video components are not supported.

The following example shows how to invoke `GNOperations.recognizeMIDFileFromFile`.

```
// Create result-ready object to receive recognition result
// ApplicationSearchResultReady must implement the GNSearchResultReady interface
ApplicationSearchResultReady searchResultReady = new ApplicationSearchResultReady();

// Invoke recognition operation with the result-ready object, a GNConfig object
// instance and an audio filename
GNOperations.recognizeMIDFileFromFile (searchResultReady, config, fileName);
```

During audio recognition, Mobile Client sends status updates to notify the application of progress.

> ⚠ Processing an audio file requires approximately 20 seconds of audio and that audio must come from the start of the track. If you want to
> later get the adjusted song position field in `GNSearchResponse`, you need to pass a `GNSampleBuffer` parameter containing a
> timestamp indicating when the sample was first collected.

GNOperations.recognizeMIDFileFromPcm

`GNOperations.recognizeMIDFileFromPcm` recognizes audio provided as a buffer of PCM (pulse-code modulation) data. This provides the application with additional flexibility: for instance, file formats not directly supported by Mobile Client can be decoded by the application to PCM and recognized via this method.

When invoked, Mobile Client reads the PCM audio data. The audio is processed and a MusicID-File fingerprint is generated. The fingerprint is then submitted to Gracenote Web Services for recognition. The result is delivered via an object that implements the `GNSearchResultReady` interface.

Status flow in PCM-based audio recognition

The following example shows how to invoke `GNOperations.recognizeMIDFileFromPcm`.

```
// Create PCM sample buffer
GNSampleBuffer sampleBuffer = new GNSampleBuffer(samples, bytesPerSample, numChannels, sampleRate);

// Create result-ready object to receive recognition result.
// ApplicationSearchResultReady must implement the GNSearchResultReady interface
ApplicationSearchResultReady searchResultReady = new ApplicationSearchResultReady();

// Invoke recognition operation with the result-ready object, a GNConfig object
// instance and the PCM buffer
GNOperations.recognizeMIDFileFromPcm (searchResultReady, config, sampleBuffer);
```

During audio recognition, Mobile Client sends status updates to notify the application of progress.

⚠ Processing an audio file requires approximately 20 seconds of audio and that audio must come from the start of the track.

AlbumID

AlbumID is a powerful and flexible recognition processing tool that can be used to provide advanced recognition of digital audio files within the user's collection. By leveraging contextual information about the audio files, AlbumID can effectively identify, group and organize a collection, providing clean and consistent metadata. It is best used for:

- Analyzing groups of media files, where the grouping of results is as important as the accuracy of the individual results
- Receiving responses that match the contextual data of an audio file, such as metadata from ID3 tags

AlbumID can use a variety of combinations of the following recognition methods and inputs:

- MusicID-File fingerprinting: Audio files in supported formats are decoded and a MusicID-File fingerprint is then generated.

- Text search and text comparison: Metadata from ID3 tags extracted from supported file formats or additional text information provided by the application are used to search for appropriate tracks and albums.
- Gracenote Identifiers: Audio files sometimes have an associated Gracenote Identifier, which Mobile Client can use for consideration during the identification process.
- Audio file name and file-system (folder) location: As music collections are often grouped by directory (Artist/Album/Track), file name and location can also used during identification.
- Audio file groupings: Files can be analyzed in groups, which allows common albums to be determined based on the tracks in the group.

> ⚠️ While commerce identifiers can be requested, AlbumID does not support the preference of a specific identifier over the actual Album a
> song came from. These preferred results can instead be obtained by using `GNOperations.recognizeMIDFileFromFile` or
> `GNOperations.recognizeMIDFileFromPcm`. See MusicID-File (see page 19) for more information.

Mobile Client provides various ways to invoke AlbumID, allowing the developer to choose between a simplified or more flexible implementation.

You can improve retrieval performance for AlbumID by retrieving enriched content in the background. For more information see Improving Retrieval Performance by Using Enriched Content URLs (see page 63).

Calling AlbumID Operations Serially

An application should call AlbumID operations serially (one at a time). An application should only call another operation after the previous operation has completed. If multiple AlbumID operations are called concurrently with many tracks (for example, 1000 tracks per AlbumID directory operation), it might impact performance on the device and might cause the application to run out of memory.

GNOperations.albumIdDirectory

`GNOperations.albumIdDirectory` takes a single directory path and identifies all of the audio files in the directory tree, including sub-directories.
This is the simplest way to invoke AlbumID.

The following code example shows how to invoke `GNOperations.albumIdDirectory`.

```
// Create result-ready object to receive recognition result.
// ApplicationSearchResultReady must implement the GNSearchResultReady interface
ApplicationSearchResultReady searchResultReady = new ApplicationSearchResultReady();

// Invoke AlbumID operation with the result-ready object, a GNConfig object
// instance and the root of the directory tree to be processed
GNOperations.albumIdDirectory(searchResultReady, config, directoryPath);
```

When invoked, this method performs the following operations:

1. Searches the directory tree and locates audio files that are supported by Gracenote MusicID-File audio decoder or AlbumID tag decoder
2. Generates MusicID-File fingerprints for files in supported formats
3. Extracts information tags from supported formats which can include artist, album and track information and Gracenote Identifiers
4. The recognition inputs collected by the above steps are combined with the file name and path and delivered to the Gracenote Service for identification; this may result in multiple queries to the Gracenote Service
5. The results of identification are delivered to the application

The behavior of `GNOperations.albumIdDirectory` can be controlled via the AlbumID configuration parameters. See AlbumID Configuration (see page 24) for more information.

GNOperations.albumIdFile

`GNOperations.albumIdFile` takes the filenames and paths of a collection of audio files and applies AlbumID identification for grouping and organizing.
This method allows targeted identification of groups of audio files, or a single audio file, utilizing all of the recognition technologies available to AlbumID.

The following code example shows how to invoke `GNOperations.albumIdFile`.

```
// Create result-ready object to receive recognition result.
// ApplicationSearchResultReady must implement the GNSearchResultReady interface
ApplicationSearchResultReady searchResultReady = new ApplicationSearchResultReady();

// Assemble a collection of audio files filename and path
ArrayList<String> filesToIdentify = new ArrayList<String>();
filesToIdentify.add("/sdcard/fileA.mp3");
filesToIdentify.add("/sdcard/fileB.mp3");

// Invoke AlbumID operation with the result-ready object, a GNConfig object
// instance and a collection of files to identify
GNOperations.albumIdFile(searchResultReady, config, filesToIdentify);
```

When invoked, this method performs the following operations:

1. Generates MusicID-File fingerprints for files in supported formats
2. Extracts information tags from supported formats which can include artist, album and track information and Gracenote Identifiers
3. The recognition inputs collected by the above steps are combined with the file name and path and delivered to the Gracenote Service for identification; this may result in multiple queries to the Gracenote Service

The behavior of `GNOperations.albumIdFile` can be controlled via the AlbumID configuration parameters. See AlbumID Configuration (see page 24) for more information.

GNOperations.albumIdList

`GNOperations.albumIdList` takes a collection of objects where each object contains the recognition inputs for an audio file. Recognition inputs are:

1. MusicID-File Fingerprint
2. Textual metadata:
    1. Artist Name
    2. Album Title
    3. Track Title
    4. Track Number
3. File name and path
4. Gracenote Identifiers

AlbumID does not require all of the above inputs to be effective; it can use only those provided to assist identification. Note this also means that text can be used if no fingerprint can be created, allowing AlbumID to be used for audio files that cannot be accessed or decoded by Mobile Client. Mobile Client does not provide a method to export Gracenote Identifiers from a file directly to the application; however, the query result contains these identifiers, which can be used for subsequent AlbumID List queries.

The following code example shows how to invoke `GNOperations.albumIdList`.

```
// Create result-ready object to receive recognition result.
// ApplicationSearchResultReady must implement the GNSearchResultReady interface
ApplicationSearchResultReady searchResultReady = new ApplicationSearchResultReady();

// Assemble a collection of GNAlbumIdAttributes objects.
// Each GNAlbumIdAttributes has the recognition inputs for a specific file.
ArrayList<GNAlbumIdAttributes> filesToIdentify = new ArrayList<GNAlbumIdAttributes>();

// Assemble the recognition inputs for individual audio files into GNAlbumIdAttributes
// instances
GNAlbumIdAttributes fileAttrib = new GNAlbumIdAttributes();
fileAttrib.setArtist("Bon Jovi");
fileAttrib.setAlbum("Keep the Faith");
fileAttrib.setFingerprintData(fingerprintString);

filesToIdentify.add(fileAttrib);

// Invoke AlbumID operation with the result-ready object, a GNConfig object
// instance and a collection of files to identify
GNOperations.albumIdList(searchResultReady, config, filesToIdentify);
```

When this method is invoked, the recognition inputs for each file are delivered to Gracenote Web Services for identification, which may result in multiple queries to Gracenote Web Services.

The behavior of `GNOperations.albumIdList` can be controlled via the AlbumID configuration parameters. See for more information.

AlbumID Configuration

The behavior of AlbumID can be controlled by appropriately configuring the `GNConfig` object provided when an AlbumID operation is invoked.

The configuration parameters are described below.

Parameter	Description	Default
`content.albumId.queryPreference.useTagData`	When `true`, AlbumID will use textual metadata when identifying an audio file.	`true`
`content.albumId.queryPreference.useFingerprint`	When `true`, AlbumID will use MusicID-Fingerprints when identifying an audio file.	`true`
`content.albumId.queryPreference.useGN_ID`	When `true`, AlbumID will use Gracenote Identifier when identifying an audio file.	`true`

AlbumID Query Load

AlbumID is capable of recognizing large music collections and in most cases will use multiple queries for recognition of tracks and retrieval of related content.

To assist with recognition Mobile Client will group audio files. The number of queries used for recognition is impacted by the number of groupings. Audio file groupings are based on:

1. The metadata of the audio files (tag data is used to group similar tracks)
2. The organization of the audio files (files can be grouped based on the directories they reside in)
3. The limit of the recognition interface

Because file groupings are impacted by track metadata and organization, it is difficult to predict how many recognition queries will be needed.

Once the audio files have been recognized, your application can access related content via the result objects. Although queries are batched in multiple groups, the responses for all the grouped queries are provided to the application at one time, at the end of the operation.

The example below provides an indication of the number of queries that are required for AlbumID. This example can be extrapolated to larger collections requiring similar content to be delivered.

AlbumID Query Load Example

Consider an audio collection organized as shown below.

Assume:

- All tracks have accurate Artist Name, Album Title, and Track Title tag data.
- No Gracenote Identifiers are known for any of the files.

```
/sdcard/Red Hot Chili Peppers/By The Way/By The Way.mp3
/sdcard/Red Hot Chili Peppers/By The Way/Universally Speaking.mp3
/sdcard/Red Hot Chili Peppers/By The Way/This Is The Place.mp3
/sdcard/Red Hot Chili Peppers/By The Way/Dosed.mp3
/sdcard/Red Hot Chili Peppers/By The Way/Don't Forget Me.mp3
/sdcard/Red Hot Chili Peppers/By The Way/...
/sdcard/Midnight Oil/10, 9, 8, 7, 6, 5, 4, 3, 2, 1/Outside World.mp3
/sdcard/Midnight Oil/10, 9, 8, 7, 6, 5, 4, 3, 2, 1/Only The Strong.mp3
/sdcard/Midnight Oil/10, 9, 8, 7, 6, 5, 4, 3, 2, 1/Short Memory.mp3
/sdcard/Midnight Oil/10, 9, 8, 7, 6, 5, 4, 3, 2, 1/Read About It.mp3
/sdcard/Midnight Oil/10, 9, 8, 7, 6, 5, 4, 3, 2, 1/Scream In Blue.mp3
/sdcard/Midnight Oil/10, 9, 8, 7, 6, 5, 4, 3, 2, 1/...
```

Using `albumIdDirectory`, the base directory `/sdcard` would be provided. The AlbumID algorithm will create two groups, one for Red Hot Chili Peppers and one for Midnight Oil. An identification query is generated for each group.

After identification is completed, the results are provided via a result-ready object. The application can then process the individual track results by processing the respective `GNSearchResponse` object.

Related content, such as Cover Art, is not included in the initial result set. When the application calls `GNSearchResponse.getCoverArt()`, a query is generated to fetch the cover art.

The application should only call `GNSearchResponse.getCoverArt()` for one track on each album. This is to avoid the same Cover Art from being fetched more than once. The application should include some intelligence to ensure this.

In this example the minimum number of queries required to identify the audio tracks and retrieve the track and album metadata and cover art for each album is four:

- Two queries for identification
- One query to retrieve Cover Art for Red Hot Chili Peppers' album By The Way
- One query to retrieve Cover Art for Midnight Oil's album 10, 9, 8, 7, 6, 5, 4, 3, 2, 1

Applications can minimize the number of queries made through correct configuration, and analysis and storage of results. Your Gracenote Professional Services representative can help design an appropriate solution for your specific needs.

Single Best Match and Multiple Matches

The audio recognition operations MusicID-Stream and MusicID-File can return a *single best match* or multiple matches. The default configuration returns a single best match, but this can be changed by configuring the `GNConfig` instance provided when the recognition operation is invoked. The multiple match configuration parameter is described below.

Parameter	Description	Default
`content.musicId.queryPreference.singleBestMatch`	When `true`, single best match is returned for MusicID-Stream and MusicID-File recognition operations. When `false`, multiple matches are returned.	`true`

When configured for a single best match, the best match is identified by comparing the characteristics of all possible matches against criteria specific to your application. By setting properties in the configuration object, you can change the criteria used in determining the best match.

When multiple matches are configured, all matches are delivered, up to a maximum of ten. The application can apply its own criteria to determine the best match and deliver it to the user. The application can also allow the user to choose the best match.

Text and Lyric Fragment Search operations always return multiple matches and AlbumID always returns a single match.

Single best match queries can be configured to return cover art in the recognition response. Cover art is not delivered with the responses containing multiple matches; however, it can be retrieved with an additional query by Gracenote Identifier (see Retrieving Related Content by Gracenote Identifier (see page 44)).

Fingerprints

Gracenote Web Services uses audio fingerprints to match audio with metadata and cover art. Mobile Client can generate audio fingerprints from any of the following sources:

- Audio captured from the microphone or other streaming audio source
- An audio file
- Pulse-code modulation (PCM-encoded) audio stored in a buffer

Generating a Fingerprint

Fingerprints are returned via a result-ready object implementing the Mobile Client interface `GNFingerprintResultReady`. Mobile Client calls the result-ready object's `GNResultReady` method when a result is generated. Your application can use this method to process the result:

```
// Result-ready object implements GNFingerprintResultReady interface
// ApplicationFingerprintResultReady must implement the GNFingerprintResultReady interface
class ApplicationFingerprintResultReady implements GNFingerprintResultReady
{
```

```
    // Provide implementation for GNResultReady method
    public void GNResultReady (GNFingerprintResult result)
    {
        // Application code to process fingerprint result
        String fingerprintData = result.getFingerprintData();
    }
}
```

The application must create the result-ready object and provide it when invoking the fingerprint generation operation:

```
// Create result-ready object to receive fingerprint result
ApplicationFingerprintResultReady fingerprintResultReady = new ApplicationFingerprintResultReady();

// Invoke fingerprint generation operation with a result-ready object and a GNConfig object
// instance
GNOperations.fingerprintMIDStreamFromMic (fingerprintResultReady, config);
```

Searching by Fingerprint

To submit a fingerprint to Gracenote Web Services for matching, use the method
`GNOperations.searchByFingerprint`:

```
// Create result-ready object to receive matching result
ApplicationSearchResultReady searchResultReady = new ApplicationSearchResultReady();

// Invoke matching operation
GNOperations.searchByFingerprint (searchResultReady, config, fingerprintData);
```

A collection of possible matches is returned as the search result. During fingerprint matching, Mobile Client sends status updates to notify the application as each of the stages shown in the figure is reached:

Status flow in fingerprint matching

Text Search

Using the method `GNOperations.searchByText`, Mobile Client can perform a text-based search in the Gracenote database for an album title, track title, and/or artist name. A combination of any of the three fields can be provided to narrow the search results:

```
// Create result-ready object to receive search result
// ApplicationSearchResultReady must implement the GNSearchResultReady interface
ApplicationSearchResultReady searchResultReady = new ApplicationSearchResultReady();

// Invoke search operation
GNOperations.searchByText (searchResultReady, config, artistName, albumTitle, trackTitle);
```

A collection of possible matches is returned as the search result; to get an exact match, use a Gracenote unique identifier instead of a text search (see "Retrieving by Track Identifier" below). During the search process, Mobile Client sends status updates to notify the application as each of the stages shown in the figure is reached:

Status flow in text search

Lyric Fragment Search

Using the method `GNOperations.searchByLyricFragment`, Mobile Client can search the Gracenote database using a lyric fragment, optionally augmented by an artist name:

```
// Create result-ready object to receive search result
// ApplicationSearchResultReady must implement the GNSearchResultReady interface
ApplicationSearchResultReady searchResultReady = new ApplicationSearchResultReady();

// Invoke search operation
GNOperations.searchByLyricFragment (searchResultReady, config, lyricFragment, artistName);
```

A collection of possible matches is returned as the search result. During the search process, Mobile Client sends status updates to notify the application as each of the stages shown in the figure is reached:

Status flow in lyric fragment search

Retrieving Related Content

Mobile Client can retrieve content related to an audio recognition result, including:

- Genre

- Mood
- Tempo
- Origin
- Era
- Artist Type
- Cover Art
- Artist Images
- Artist Biographies
- Album Reviews

You can retrieve related content from an audio recognition operation result or via a Gracenote unique album or track identifier.

> For more information on using images returned by Mobile Client see Image Resizing Best Practice (see page 56).

Genre

Mobile Client returns album and track level genre descriptors with a recognition or search result. Genre descriptors can be displayed to the end-user or can be used to categorize music in the user's collection for organization, navigation, or playlist generation.

Genre Levels

Gracenote has multiple levels of genre descriptors. Each level describes the related music with a different amount of detail and granularity.

Two options for genre are available: `DEFAULT` and `EXTENDED`. The `DEFAULT` option returns a single, default genre descriptor in the result. The `EXTENDED` option returns multiple levels of genre descriptors in the result.

The recommended option to use depends upon the application, its use cases, and the target audience. Some applications might want to use the `DEFAULT` option, as it is the simplest and contains the most commonly known genres. Other applications might want to provide a richer genre experience via the multiple levels in the `EXTENDED` option. Applications might also use the `EXTENDED` option to give their users a choice of which genre level is used.

A Mobile Client operation can be configured for one of these options via the `GNConfig` object provided when the operation is invoked. These parameters are described below.

Parameter	Description	Default
`content.genre.level`	Sets the option for the genre descriptors returned; DEFAULT and EXTENDED	`DEFAULT`

Genre Localization

Gracenote provides a global solution and Mobile Client can be configured to deliver genres specific to a supported country and in a supported language. The country and language can be configured by setting the appropriate parameters in the `GNConfig` object provided when invoking a `GNOperation`. These parameters are described below.

Parameter	Description	Default
`content.country`	Specifies the country of the delivered genre descriptors (using ISO county code)	`null` (as of version 2.5.8, `USA` before)
`content.lang`	Specifies the language of the delivered genre descriptors (using ISO lang code)	`""`

See Localization and Internationalization (see page 48) for more information.

Accessing Genres

Genre data is delivered via a result-ready object in a `GNDescriptor` instance. Multiple genre levels can be delivered for a single album or track, so a collection of `GNDescriptor` objects is delivered.

> ⚠ When track-level genre descriptors are requested, Mobile Client returns them, if they are available. If track-level descriptors are not available, album-level descriptors are returned instead.

The delivered genre levels are stored in an array of `GNDescriptor` objects. The order of the `GNDescriptor` objects in the collection is representative of their level.

Genres are defined by a descriptor and an identifier. The descriptor is a word or phrase describing the genre. The descriptor string should only be used to display to the end user. Because descriptor names and user language settings are subject to change, the identifier is best suited for dealing with genre associations in a programmatic fashion.

The `GNDescriptor` object is fully described in the data dictionary, see GNDescriptor (see page 53).

The following code snippet shows how genres can be retrieved from a result-ready object.

```
// Define result-ready object to extract related content from a search result
class ApplicationSearchResultReady implements GNSearchResultReady
{
    // Provide implementation for GNResultReady method
    public void GNResultReady (GNSearchResult result)
    {
        // Get best response from search result
        GNSearchResponse bestResponse = result.getBestResponse();

        if (bestResponse != null)
        {
            // Extract track level genres collection
            GNDescriptor[] trackLevelGenres = bestResponse.getTrackGenre();
            for(GNDescriptor trackGenre: trackLevelGenres)
            {
                // Extract the genre information for each genre level returned
                String genreDescriptor = trackGenre.getData();
```

```
            String genreId = trackGenre.getId();
            // Process genre information appropriately
        }

        // Extract album level genres collection
        GNDescriptor[] albumLevelGenres = bestResponse.getAlbumGenre();
        for(GNDescriptor albumGenre: albumLevelGenres)
        {
            // Extract the genre information for each genre level returned
            String genreDescriptor = albumGenre.getData();
            String genreId = albumGenre.getId();
            // Process genre information appropriately
        }
    }
  }
}
```

Mood

Mobile Client can return track-level mood descriptors with a recognition or search result. Mood descriptors can be displayed to the end-user or can be used to categorize music in the user's collection for organization, navigation, or playlist generation.

Your application must be entitled to retrieve mood data. To entitle your application, please contact your Gracenote Professional Services representative.

A Mobile Client operation can be configured to include mood descriptors in its response, via the `GNConfig` object provided when the operation is invoked. These parameters are described below.

Parameter	Description	Default
`content.mood`	Set to `true` to enable retrieval of mood descriptors	`false`
`content.mood.level`	Sets the option for the mood descriptors returned; `DEFAULT` and `EXTENDED`	`DEFAULT`

Mood Levels

Gracenote has multiple levels of mood descriptors. Each level describes the related music with a different amount of detail and granularity.

Two options for mood are available: `DEFAULT` and `EXTENDED`. The `DEFAULT` option returns a single, default mood descriptor in the result. The `EXTENDED` option returns multiple levels of mood descriptors in the result.

The recommended option to use depends upon the application, its use cases, and the target audience. Some applications might want to use the `DEFAULT` option, as it is the simplest and contains the most commonly known moods. Other applications might want to provide a richer experience via the `EXTENDED` option. Applications might also use the the `EXTENDED` option to give their users a choice of which mood level is used.

Mood Localization

Gracenote provides a global solution and Mobile Client can be configured to deliver moods in a supported language. The language can be configured by setting the appropriate parameter in the `GNConfig` object provided when invoking a `GNOperation`. These parameters are described below.

Parameter	Description	Default
`content.lang`	Specifies the language of the delivered mood descriptors	""

See Localization and Internationalization (see page 48) for more information.

Accessing Moods

Mood data is delivered via a result-ready object in a `GNDescriptor` instance. Multiple mood levels may be delivered for a single track, so a collection of `GNDescriptor` objects is delivered.

The delivered mood levels are stored in an array of `GNDescriptor` objects. The order of the `GNDescriptor` objects in the collection is representative of their level.

Moods are defined by a descriptor and an identifier. The descriptor is a word or phrase describing the mood. The descriptor string should only be used to display to the end user. Because descriptor names and user language settings are subject to change, the identifier is best suited for dealing with mood associations in a programmatic fashion.

The `GNDescriptor` object is fully described in the data dictionary, see GNDescriptor (see page 53).

The following code snippet shows how mood data can be retrieved from a result-ready object.

```
// Define result-ready object to extract related content from a search result
class ApplicationSearchResultReady implements GNSearchResultReady
{
// Provide implementation for GNResultReady method
public void GNResultReady (GNSearchResult result)
{
// Get best response from search result
GNSearchResponse bestResponse = result.getBestResponse();

if (bestResponse != null)
{
// Extract track level mood collection
GNDescriptor[] trackLevelMoods = bestResponse.getMood();
for(GNDescriptor trackMood: trackLevelMoods){
// Extract the mood information for each mood level returned
String moodDescriptor = trackMood.getData();
String moodId = trackMood.getId();
// Process mood information appropriately
}
```

```
    }
  }
}
```

Tempo

Mobile Client can return track-level tempo data with a recognition or search result. Tempo data can be displayed to the end-user or can be used to categorize music in the user's collection for organization, navigation, or playlist generation.

Your application must be entitled to retrieve tempo data. To entitle your application, please contact your Gracenote Professional Services representative.

A Mobile Client operation can be configured to include tempo data in its response, via the `GNConfig` object provided when the operation is invoked. These parameters are described below.

Parameter	Description	Default
`content.tempo`	Set to `true` to enable retrieval of tempo descriptors	`false`
`content.tempo.level`	Sets the option for the tempo descriptors returned; `DEFAULT` and `EXTENDED`	`DEFAULT`

Tempo Levels

Gracenote has multiple levels of tempo descriptors. Each level describes the related music with a different amount of detail and granularity.

Two options for tempo are available: `DEFAULT` and `EXTENDED`. The `DEFAULT` option returns a single, default tempo descriptor in the result. The `EXTENDED` option returns multiple levels of tempo descriptors in the result.

The recommended option to use depends upon the application, its use cases, and the target audience. Some applications might want to use the `DEFAULT` option, and other applications might want to provide a richer experience via the `EXTENDED` option. Applications might also use the `EXTENDED` option to give their users a choice of which tempo level is used.

Tempo Localization

Gracenote provides a global solution and Mobile Client can be configured to deliver tempo in a supported language. The language can be configured by setting the appropriate parameter in the `GNConfig` object provided when invoking a `GNOperation`. These parameters are described below.

Parameter	Description	Default
`content.lang`	Specifies the language of the delivered tempo descriptors	`""`

See Localization and Internationalization (see page 48) for more information.

Accessing Tempo Data

Tempo data is delivered via a result-ready object in a `GNDescriptor` instance. Multiple tempo levels may be delivered for a single track, so a collection of `GNDescriptor` objects is delivered.

The delivered tempo levels are stored in an array of `GNDescriptor` objects. The order of the `GNDescriptor` objects in the collection is representative of their level.

Tempos are defined by a descriptor and an identifier. The descriptor is a word or phrase describing the tempo. The descriptor string should only be used to display to the end user. Because descriptor names and user language settings are subject to change, the identifier is best suited for dealing with tempo associations in a programmatic fashion.

The `GNDescriptor` object is fully described in the data dictionary, see GNDescriptor (see page 53).

The following code snippet shows how tempo data can be retrieved from a result-ready object.

```
// Define result-ready object to extract related content from a search result
class ApplicationSearchResultReady implements GNSearchResultReady
{
// Provide implementation for GNResultReady method
public void GNResultReady (GNSearchResult result)
{
// Get best response from search result
GNSearchResponse bestResponse = result.getBestResponse();

if (bestResponse != null)
{
// Extract track level tempo collection
GNDescriptor[] trackLevelTempos = bestResponse.getTempo();
for(GNDescriptor trackTempo: trackLevelTempos){
// Extract the tempo information for each tempo level returned
String tempoDescriptor = trackTempo.getData();
String tempoId = trackTempo.getId();
// Process tempo information appropriately
}
}
}
}
```

Origin, Era, and Artist Type

Mobile Client can return origin, era, and artist type data with a recognition or search result. Origin data gives the geographic location most strongly associated with the artist. Era data gives the time period most strongly associated with the artist. Artist type data gives the gender and composition (solo, duo, group) of the artist. Origin, era, and artist type data can be displayed to the end-user or can be used to categorize music in the user's collection for organization, navigation, or playlist generation.

A Mobile Client operation can be configured to include origin, era, and artist type data in its response, via the GNConfig object provided when the operation is invoked. These parameters are described below.

Parameter	Description	Default
`content.origin`	Set to `true` to enable retrieval of origin descriptors	`false`
`content.era`	Set to `true` to enable retrieval of era descriptors	`false`
`content.artistType`	Set to `true` to enable retrieval of artist type descriptors	`false`

⚠️ Origin, era, and artist type data is delivered as a bundle. If the value of any of the parameters is set to `TRUE`, all three types of descriptors are delivered. To turn off delivery of the descriptors, all three parameters must be set to `FALSE`.

Origin, Era, and Artist Type Levels

Gracenote has multiple levels of origin, era, and artist type descriptors. Each level describes the related music with a different amount of detail and granularity. The following GNConfig parameters set the option for the origin, era, and artist type descriptors returned:

`content.origin.level`	Sets the option for the origin descriptors returned; `DEFAULT` and `EXTENDED`	`DEFAULT`
`content.era.level`	Sets the option for the era descriptors returned; `DEFAULT` and `EXTENDED`	`DEFAULT`
`content.artistType.level`	Sets the option for the artist type descriptors returned; `DEFAULT` and `EXTENDED`	`DEFAULT`

Two options for origin, era, and artist type descriptors are available: `DEFAULT` and `EXTENDED`. The `DEFAULT` option returns a single, default descriptor in the result. The `EXTENDED` option returns multiple levels of descriptors in the result.

The recommended option to use depends upon the application, its use cases, and the target audience. Some applications might want to use the DEFAULT option, and other applications might want to provide a richer experience via the EXTENDED option. Applications might also use the EXTENDED option to give their users a choice of which level is used.

Origin, Era, and Artist Type Localization

Gracenote provides a global solution and Mobile Client can be configured to deliver origin, era, and artist type in a supported language. The language can be configured by setting the appropriate parameter in the `GNConfig` object provided when invoking a `GNOperation`. The parameter is described below.

Parameter	Description	Default
`content.lang`	Specifies the language of the delivered tempo descriptors	""

See for more information.

Accessing Origin, Era, and Artist Type Data

Origin, era, and artist type data is delivered via a result-ready object in a GNDescriptor instance. Multiple levels may be delivered, so a collection of GNDescriptor objects is delivered.

The delivered levels are stored in an array of GNDescriptor objects. The order of the GNDescriptor objects in the collection is representative of their level.

Origin, era, and artist type are defined by a descriptor and an identifier. The descriptor is a word or phrase describing the origin, era, or artist type. The descriptor string should only be used to display to the end user. Because descriptor names and user language settings are subject to change, the identifier is best suited for dealing with origin, era, and artist type associations in a programmatic fashion.

The `GNDescriptor` object is fully described in the data dictionary, see GNDescriptor (see page 53).

The following code snippet shows how origin, era, and artist type data can be retrieved from a result-ready object.

```
if (bestResponse != null)
{
// Extract artist origin value(s)
GNDescriptor[] originDescriptorArray = bestResponse.getOrigin();
for(GNDescriptor originDesciptor: originDescriptorArray){
// Extract the origin information for each origin level returned
String origin = originDescriptor.getData();
String originId = originDescriptor.getId();
// Process origin information appropriately
}

// Extract artist era value(s)
GNDescriptor[] eraDescriptorArray = bestResponse.getEra();
for(GNDescriptor eraDesciptor: eraDescriptorArray){
// Extract the era information for each era level returned
String era = eraDescriptor.getData();
String eraId = eraDescriptor.getId();
// Process era information appropriately
}

// Extract artist type value(s)
GNDescriptor[] artistTypeDescriptorArray = bestResponse.getArtistType();
for(GNDescriptor artistTypeDesciptor: artistTypeDescriptorArray){
// Extract the artist type information for each artist type level returned
String artistType = artistTypeDescriptor.getData();
String artistTypeId = artistTypeDescriptor.getId();
// Process artist type information appropriately
}
}
}
}
```

Cover Art

Mobile Client can return album Cover Art with a recognition or search result. Cover Art can be used effectively to enrich the user experience.

Note that only the cover art URL is returned with the result. To get the actual image data, your app needs to invoke the `GNCoverArt.data` accessor method, which should be done in a background thread. See the Sample App for an example of getting cover art data in a background thread.

Your application must be entitled to retrieve Cover Art. To entitle your application, please contact your Gracenote Professional Services representative.

A Mobile Client operation can be configured to include Cover Art in its response, via the `GNConfig` object provided when the operation is invoked. These parameters are described below.

Parameter	Description	Default
`content.coverArt`	Set to `true` to enable retrieval of cover art	`false` (as of version 2.5.8)
`content.coverArt.sizePreference`	Comma-separated list of cover art size preferences. Also specifies size preference for artist images.	`"SMALL,MEDIUM,THUMBNAIL,LARGE,XLARGE"`

The Cover Art size preference takes a comma-separated list of the preferred Cover Art sizes, in order of preference. Mobile Client will return the first Cover Art image that matches a size in the preference list. If a size is not in the preference list, it will not be returned.

> ⚠ If an invalid size preference is included in the list, Mobile Client ignores the list and uses the default list instead.

For more information on using images returned by Mobile Client see Image Resizing Best Practice (see page 56).

You can improve retrieval performance for some operations by retrieving cover art in the background. For more information see Improving Retrieval Performance by Using Enriched Content URLs (see page 63).

Genre Cover Art

In cases where Gracenote does not have cover art for a particular album, genre-themed artwork is instead returned in a response. This option can be disabled on a per-query basis by setting the appropriate `GNConfig` parameter as shown below.

Parameter	Description	Default
`content.coverArt.genreCoverArt`	Set to `true` to receive genre cover art when album cover art is not available	`true`

> ⚠ To receive genre art, both `content.coverArt.genreCoverArt` and `content.coverArt` **must** be set to `true`. Your application also **must** be entitled for cover art.

The size of the Genre Cover Art returned is governed by the Cover Art size preference `GNConfig` parameter described above.

For more information on using images returned by Mobile Client see Image Resizing Best Practice (see page 56).

Accessing Cover Art

Cover Art is delivered via a result-ready object as a `GNCoverArt` object. Cover Art is provided as a raw stream that can be converted to an Android `Drawable` object. Once a `Drawable` object is created, the Cover Art can be easily displayed to the user via standard Android mechanisms.

The `GNCoverArt` object is fully described in the data dictionary, see GNCoverArt (see page 53).

The following code snippet shows how Cover Art data can be retrieved from a result-ready object.

```
// Define result-ready object to extract related content from a search result
class ApplicationSearchResultReady implements GNSearchResultReady
{
// Provide implementation for GNResultReady method
public void GNResultReady (GNSearchResult result)
{
// Get best response from search result
GNSearchResponse bestResponse = result.getBestResponse();

if (bestResponse != null)
{
GNCoverArt coverArt = bestResponse.getCoverArt();
if( coverArt != null ){
// Convert the cover art to an Android Drawable
ByteArrayInputStream is = new ByteArrayInputStream(coverArt().getData());
Drawable coverArt = Drawable.createFromStream(is, "src");
// Process cover art Drawable
}
}
}
}
```

Artist Images

Mobile Client can return an Artist Image with a recognition or search result. Artist Images can be used effectively to enrich the user experience.

Your application must be entitled to retrieve Artist Images. To entitle your application, please contact your Gracenote Professional Services representative.

A Mobile Client operation can be configured to include an Artist Image in its response, via the `GNConfig` object provided when the operation is invoked. These parameters are described below.

Parameter	Description	Default
content.contributor.images	Set to `true` to enable retrieval of artist images	false
content.coverArt.sizePreference	Comma-separated list of artist image size preferences. Also specifies size preference for cover art images.	"SMALL,MEDIUM,THUMBNAIL,LARGE,XLARGE"

The Artist Image size preference takes a comma-separated list of the preferred Artist Image sizes, in order of preference. Mobile Client returns the first Artist Image that matches a size in the preference list. If a size is not in the preference list, it will not be returned.

For more information on using images returned by Mobile Client see Image Resizing Best Practice (see page 56).

You can improve retrieval performance for some operations by retrieving artist images in the background. For more information see Improving Retrieval Performance by Using Enriched Content URLs (see page 63).

Accessing Artist Images

Artist Images are delivered via a result-ready object as a `byte[]`. The raw data can be converted to an Android `Drawable` object. Once a `Drawable` object is created, the image can be easily displayed to the user via standard Android mechanisms.

The following code snippet shows how artist images can be retrieved from a result-ready object.

```
// Define result-ready object to extract related content from a search result
class ApplicationSearchResultReady implements GNSearchResultReady
{
// Provide implementation for GNResultReady method
public void GNResultReady (GNSearchResult result)
{
// Get best response from search result
GNSearchResponse bestResponse = result.getBestResponse();

if (bestResponse != null)
{
GNImage artistImage = bestResponse.getContributorImage();
byte[] artistImageData = artistImage.getData();
if( artistImageData != null ){
// Convert the image data to an Android Drawable
ByteArrayInputStream is = new ByteArrayInputStream(artistImageData);
Drawable artistImage = Drawable.createFromStream(is, "src");
// Process artist image Drawable
}
}
}
}
```

Artist Biographies

Mobile Client can return an Artist Biography with a recognition or search result. Artist Biographies can be used effectively to enrich the user experience.

Your application must be entitled to retrieve Artist Biographies. To entitle your application, please contact your Gracenote Professional Services representative.

A Mobile Client operation can be configured to include an Artist Biography in its response, via the `GNConfig` object provided when the operation is invoked. These parameters are described below.

Parameter	Description	Default
`content.contributor.biography`	Set to `true` to enable retrieval of artist biographies	`false`

You can improve retrieval performance for some operations by retrieving artist biographies in the background. For more information see .

Accessing Artist Biographies

Artist Biographies are delivered via a result-ready object as a collection of `String` objects, where each `String` represents an individual paragraph of the biography.

The following code snippet shows how an artist biography can be retrieved from a result-ready object.

```
// Define result-ready object to extract related content from a search result
class ApplicationSearchResultReady implements GNSearchResultReady
{
// Provide implementation for GNResultReady method
public void GNResultReady (GNSearchResult result)
{
// Get best response from search result
GNSearchResponse bestResponse = result.getBestResponse();

if (bestResponse != null)
{
// Extract the artist biography
String[] bio = bestResponse.getArtistBiography();
// Process artist biography information appropriately
}
}
}
```

Album Reviews

Mobile Client can return an Album Review with a recognition or search result. Album Reviews can be used effectively to enrich the user experience.

Your application must be entitled to retrieve Album Reviews. To entitle your application, please contact your Gracenote Professional Services representative.

A Mobile Client operation can be configured to include Album Reviews in its response, via the `GNConfig` object provided when the operation is invoked. These parameters are described below.

Parameter	Description	Default
`content.review`	Set to `true` to enable retrieval of album reviews	`false`

You can improve retrieval performance for some operations by retrieving album reviews in the background. For more information see .

Accessing Album Reviews

Album Reviews are delivered via a result-ready object as a collection of `String` objects, where each `String` represents an individual paragraph of the review.

The following code snippet shows how an album review can be retrieved from a result-ready object.

```
// Define result-ready object to extract related content from recognition result
class ApplicationSearchResultReady implements GNSearchResultReady
{
// Provide implementation for GNResultReady method
public void GNResultReady (GNSearchResult result)
{
// Get best response from search result
GNSearchResponse bestResponse = result.getBestResponse();

if (bestResponse != null)
{
// Extract the artist biography
String[] review = bestResponse.getAlbumReview();
// Process album review information appropriately
}
}
}
```

Retrieval Methods

Related content can be retrieved from the single best match returned by an audio recognition operation or via a Gracenote unique album or track identifier.

Retrieving Related Content for a Single Best Match

A single best match result returned by an audio recognition operation can contain related content, if the `GNConfig` object provided when the operation is invoked is appropriately configured. This mechanism can be used to deliver all of the desired related content in a single `GNOperations` call.

The following code example illustrates how to configure the configuration object, initiate a recognition event, and retrieve the related content from the returned result:

```
// Set configuration properties
config.setProperty ("content.genre", "true");
config.setProperty ("content.mood", "true");
config.setProperty ("content.tempo", "true");
config.setProperty ("content.origin", "true");
config.setProperty ("content.era", "true");
config.setProperty ("content.artistType", "true");
config.setProperty ("content.coverArt", "true");
config.setProperty ("content.contributor.images", "true");
config.setProperty ("content.contributor.biography", "true");
config.setProperty ("content.review", "true");


// Define result-ready object to extract related content from recognition result
class ApplicationSearchResultReady implements GNSearchResultReady
{
// Provide implementation for GNResultReady method
public void GNResultReady (GNSearchResult result)
{
// Get best response from search result
GNSearchResponse bestResponse = result.getBestResponse();

if (bestResponse != null)
{
// Extract related content from response
GNDescriptor[] trackGenres = bestResponse.getTrackGenre();
GNDescriptor[] albumGenres = bestResponse.getAlbumGenre();
GNDescriptor[] trackMoods = bestResponse.getMood();
GNDescriptor[] trackTempos = bestResponse.getTempo();
GNDescriptor[] artistOrigins = bestResponse.getOrigin();
GNDescriptor[] artistEras = bestResponse.getEra();
GNDescriptor[] artistTypes = bestResponse.getArtistType();
GNCoverArt coverArt = bestResponse.getCoverArt();
GNImage artistImage = bestResponse.getContributorImage();
String[] artistBiography = bestResponse.getArtistBiography();
String[] albumReview = bestResponse.getAlbumReview();

// Display related content as desired
}
}
}


// Use configuration object when invoking a recognition operation
void recognizeFromMic (GNConfig config)
{
// Create result-ready object to receive retrieval result and extract related content
ApplicationSearchResultReady searchResultReady = new ApplicationSearchResultReady();
```

```
// Invoke recognition operation
GNOperations.recognizeMIDStreamFromMic (searchResultReady, config);
}
```

> For subsequent operations after an initial identification, retrieving related content from a Gracenote album or track identifier is more efficient than repeating the full recognition process. To minimize response time, extract and store the Gracenote Identifier after the initial recognition operation and use it for subsequent retrieval operations, as shown in the next section.

Retrieving Related Content by Gracenote Identifier

Gracenote unique album and track identifiers can be obtained from any Gracenote product, enabling Mobile Client to be easily integrated into a larger music identification system. The following code example shows how to obtain a Gracenote Identifier for a specific album and use it to retrieve related content; the same technique can be used to retrieve related content for a track identifier instead of an album identifier, by using the methods `GNSearchResponse.getTrackId` and `GNOperations.fetchByTrackId` instead of `GNSearchResponse.getAlbumId` and `GNOperations.fetchByAlbumId`:

```
// Define result-ready object to extract related content from recognition result by album identifier
class ApplicationGetGnIdResultReady implements GNSearchResultReady
{
// Provide implementation for GNResultReady method
public void GNResultReady (GNSearchResult result)
{
// Get best response from search result
GNSearchResponse bestResponse = result.getBestResponse();

// Extract album identifier and store for later use
gnAlbumID = bestResponse.getAlbumID();

// Retrieve related content using album identifier
fetchRelatedContent (config, gnAlbumID);
}
}


// Define result-ready object to extract related content from recognition result
class ApplicationFetchRelatedContent implements GNSearchResultReady
{
public void GNResultReady (GNSearchResult result)
{
// Get best response from search result
GNSearchResponse bestResponse = result.getBestResponse();

// Extract related content from response
if (bestResponse != null)
{
GNDescriptor[] trackGenres = bestResponse.getTrackGenre();
GNDescriptor[] albumGenres = bestResponse.getAlbumGenre();
GNDescriptor[] trackMoods = bestResponse.getMood();
GNDescriptor[] trackTempos = bestResponse.getTempo();
GNDescriptor[] artistOrigins = bestResponse.getOrigin();
```

```
GNDescriptor[] artistEras = bestResponse.getEra();
GNDescriptor[] artistTypes = bestResponse.getArtistType();
GNCoverArt coverArt = bestResponse.getCoverArt();
GNImage artistImage = bestResponse.getContributorImage();
String[] artistBiography = bestResponse.getArtistBiography();
String[] albumReview = bestResponse.getAlbumReview();


// Display the related content as desired
}
}
}


// Initial recognition request; result will contain a Gracenote album identifier
void recognizeFromMic (GNConfig config)
{
// Create result-ready object to receive search result and extract and store track identifier and related
content
ApplicationGetGnIdResultReady getGnIdResultReady = new ApplicationGetGnIdResultReady();

// Invoke recognition operation
GNOperations.recognizeMIDStreamFromMic (getGnIdResultReady, config);
}


// Later retrieval request using previously stored album identifier
void fetchRelatedContent (GNConfig config, String albumId)
{
// Set configuration properties
config.setProperty ("content.genre", "true");
config.setProperty ("content.mood", "true");
config.setProperty ("content.tempo", "true");
config.setProperty ("content.origin", "true");
config.setProperty ("content.era", "true");
config.setProperty ("content.artistType", "true");
config.setProperty ("content.coverArt", "true");
config.setProperty ("content.contributor.images", "true");
config.setProperty ("content.contributor.biography", "true");
config.setProperty ("content.review", "true");

// Create result-ready object to receive retrieval result
ApplicationFetchRelatedContent fetchRelatedContent = new ApplicationFetchRelatedContent();

// Invoke retrieval operation using album identifier
GNOperations.fetchByAlbumId (fetchRelatedContent, config, albumId);
}
```

During the retrieval process, Mobile Client sends status updates to notify the application as each stage shown in the figure is reached:
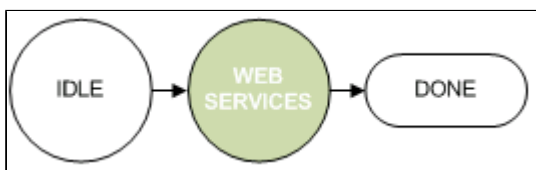
Status flow in related content retrieval by Gracenote Identifier

Track and Album Identifiers can be retrieved from any of Gracenote's products, enabling Mobile Client to be easily integrated into a larger music identification system.

Retrieving Link Data

When an application is appropriately entitled, Mobile Client returns Link identifiers for all responses provided in a result-ready object instance. The identifiers may be sourced from a third party (such as Amazon.com), or they may be specific to your organization. You can use such Link data to (for example) redirect the user to a site, such as Amazon, where they can purchase the matched track, or as a key into your own data catalog. Consult your Gracenote Professional Services representative for further information about the Link product and typical uses in commerce, or to entitle your application for Link access.

```
class ApplicationSearchResultReadyObject implements GNSearchResultReady{
public void GNResultReady(GNSearchResult result) {
// Application code to process the search result
GNSearchResponse response[] = result.getResponses();
if( response.length != 0 ){
GNLinkData linkData[] = response[0].getAlbumLinkData();
if( linkData.length != 0 ){
// User code to use link data
}
}
}
}

void recognizeFromMic( GNConfig config ){
ApplicationSearchResultReadyObject searchResultReadyObject = new ApplicationSearchResultReadyObject();
GNOperations.recognizeMIDStreamFromMic(searchResultReadyObject,config);
}
```

You can configure audio recognition events to prefer results containing Link identifiers from a specific source. Set the content.link.preferredSource property of the `GNConfig` object used when invoking the audio recognition method, as shown in this example:

```
GNConfig config = GNConfig.init("12345678-ABCDEFGHIJKLMNOPQRSTUVWXYZ012345",applicationContext);

config.setProperty("content.link.preferredSource","Amazon");

GNOperations.recognizeMIDStreamFromMic(searchResultReadyObject,config);
```

Gracenote Web Services will then return the best available match containing a Link identifier from that source, if available. Note, however, that the single best match returned is not necessarily guaranteed to contain a Link identifier from the preferred source, if none of the available matches contains one from that source.

⚠️

> While commerce identifiers can be requested, AlbumID does not support the preference of a specific identifier over the actual Album a song came from. These preferred results can instead be obtained by using `GNOperations.recognizeMIDFileFromFile` or `GNOperations.recognizeMIDFileFromPcm`. See for more information.

Status Change Updates

When performing an operation, Mobile Client sends status updates to the application via the `GNOperationStatusChanged` interface.

To receive status changed updates, the application must create a class that implements a result-ready interface and the `GNOperationStatusChanged` interface. An instance of the object must be provided when the operation is invoked.

Your app can use status changed updates to keep the user notified of operation progress. The percent completed is also provided but, currently, only for microphone recorded audio.

> ⚠ The callbacks for handling status changes and results are called on the main thread. As a best practice, your app should keep the main thread readily available for this purpose and run any other time-consuming tasks in the background.

```
class SearchResultsStatusReceiver implements GNSearchResultReady,GNOperationStatusChanged{
public void GNResultReady( GNSearchResult result ){
// User code to process search result
}

public void GNStatusChanged( GNStatus status ){
// User code to process status changed update

if( status.getStatus() == GNStatusEnum.LISTENING){
int percentDone = status.getPercentDone();
// Display percent completed to the user
}
}
};

void recognizeFromMic( GNConfig config ){
SearchResultsStatusReceiver searchResultStatusReceiverObject = new SearchResultsStatusReceiver();
GNOperations.recognizeMIDStreamFromMic(searchResultReadyObject,config);
}
```

Canceling an Operation

Mobile Client supports canceling a running operation. To cancel an operation, the application must call the `GNOperations.cancel` method with the same result-ready object instance that was provided to the operation when it was invoked. Mobile Client uses the result-ready object instance to identify which operation to cancel.

```
class RecognizeFromMic{
private SearchResultsStatusReceiver searchResultStatusReceiverObject;
```

```
void recognizeFromMic( GNConfig config ){
searchResultStatusReceiverObject = new SearchResultsStatusReceiver();
GNOperations.recognizeMIDStreamFromMic(searchResultReadyObject,config);
}

void cancelRecognize(){
GNOperations.cancel(searchResultStatusReceiverObject);
}
}
```

When `GNOperations.cancel` is called, the associated operation is stopped. `cancel` is a non-blocking call and, once invoked, your application can continue with other operations and calls. Your application will not receive any additional information about the cancelled operation.

The `GNRecognizeStream.stopRecognizeSession` method cancels the IDNow operation and stops the session as well. To cancel the IDNow operation without stopping the session, call `GNRecognizeStream.cancelIdNow`.

For more information on using cancel, contact your Gracenote Professional Services representative.

Localization and Internationalization

Gracenote provides a global solution and Mobile Client can be configured to deliver metadata that is specific to a supported region or a supported language. A country or language can be configured by setting the appropriate `GNConfig` parameters. The parameters are described below.

Parameter	Description
content.country	Specifies the country of the delivered metadata (default is `null` as of version 2.5.8, `USA` before)
content.lang	Specifies the language of the delivered metadata

The specified country and language can affect the delivered metadata in various ways:

1. When a supported country is specified, the genre descriptors for that country are delivered
2. When a supported language is specified, album and track metadata are delivered in that language, if available
3. When a supported language is specified, genre, mood, tempo, origin, era, and artist type descriptors are delivered in that language, if available

For a complete list of supported languages see the *Mobile Client Reference Guide*.
All country codes specified by *ISO 3166-1 alpha-3* are supported.

The Android Developers Guide also provides information on supporting localization and internationalization.

Debug Logging

Mobile Client can be configured to generate a debug log and PCM for debugging. To turn on debugging, set the debugEnabled property to "1", "true" or "True" in a `GNConfig` object instance, and provide that instance when invoking an operation.

Only operations that are passed a GNConfig object instance with debugging enabled will generate logging output.
The `recognizeMIDStreamFromMic` and `fingerprintMIDStreamFromMic` operations with debugging enabled will generate recorded.pcm.

The default location of the generated log file is `/sdcard/gracenote/debug.log` and PCM file is `/sdcard/gracenote/recorded.pcm`. The location of the generated files can be changed by setting the debugLog property of a `GNConfig` object instance.

When choosing a location for the debug files ensure:

1. The location is specified
2. The location exists
3. The application has write access to the location
4. The media that will store the debug files has enough available memory

```
config.setProperty("debugEnabled", "1");
config.setProperty("debugLog", "/sdcard/debugDir");
```

The debug files feature is provided for development only. Production applications cannot have debugging enabled or provide an option to enable it.

During validation Gracenote ensures debug files generation are not and can not be enabled by the application.

Data Dictionary

This section describes the different data fields returned from Mobile Client. Two different types of results can be generated by Mobile Client, a music search result and a fingerprint creation result. Both result types are described in the sections below.

Search Result

A Mobile Client search result returns a variety of metadata fields that can be used to enrich the user experience. The following sections provide a description of each field and the hierarchy in which the fields are delivered to your application.

```
java.lang.Object
  :
  '--com.gracenote.mmid.MobileSDK.GNResult
      :
      '-- com.gracenote.mmid.MobileSDK.GNFingerprintResult
          :
          '-- com.gracenote.mmid.MobileSDK.GNSearchResult
              ├──com.gracenote.mmid.MobileSDK.GNSearchResponse[]
              └──com.gracenote.mmid.MobileSDK.GNLinkData[]

      ····· Inherits from
      ——— Contains
```

GNSearchResult

Contains the result(s) of a search operation.

Field	Description	Type	Accessor	Comments
Best Response	Album that contains the track that is the best match of the search operation	GNSearchResponse	getBestresponse	
Responses	Collection of albums that contain tracks that were matched by the search operation	GNSearchResponse[]	getResponses	

GNSearchResponse

Describes the metadata fields for an album and the track on that album that was matched by the search operation.

Field	Description	Type	Accessor	Comments
Album Artist Name	Name of the album level artist	String	getAlbumArtist	
Album Artist Name Betsumei	Japanese alternate names and pronunciations for album level artist name	String	getAlbumArtistBetsumei	
Album Artist Name Yomi	Japanese phonetic representation of album level artist name	String	getAlbumArtistYomi	
Album Genre	Album level genre descriptors	ArrayList<GNDescriptor>	getAlbumGenre	
Album ID		String	getAlbumId	

	Gracenote unique identifier for the album			
Album Link Data	Link identifiers related to the album	GNLinkData[]	getAlbumLinkData	This could be null
Album Release Year	Year the album was released in	String	getAlbumReleaseYear	
Album Review	Reviews on album	String[]	getAlbumReview	This ranges from zero to several paragraphs.
Album Track Count	Number of tracks on the album	int	getAlbumTrackCount	-1 if no track
Artist Biography	Artist biography	String[]	getArtistBiography	This ranges from zero to several paragraphs.
Arist Era	Era descriptors	ArrayList<GNDescriptor>	getEra	
Artist Image	Track/album artist image	GNImage	getContributorImage	
Artist Origin	Origin descriptors	ArrayList<GNDescriptor>	getOrigin	
Artist Type	Artist type descriptors	ArrayList<GNDescriptor>	getArtistType	
Cover Art	Album cover art	GNCoverArt	getCoverArt	When cover art is not available, genre art may be returned (based on availability).
Track Artist Name	Name of the track level artist	String	getArtist	
Track Artist Name Betsumei	Japanese alternate names and pronunciations for track level artist name	String	getArtistBetsumei	
Track Artist Name Yomi	Japanese phonetic representation of track level artist name	String	getArtistYomi	
Track Duration	Track length	String	getSongDuration	
Track Genre	Track level genre descriptors	ArrayList<GNDescriptor>	getTrackGenre	If track-level descriptors are not available, album-level descriptors are returned instead.
Track ID	Gracenote unique identifer for the track	String	getTrackId	This could be null in the case of Lyric_search
		GNLinkData[]	getTrackLinkData	This could be null

Track Link Data	Link identifiers related to the track			
Track Match Position	Indicates at what timeslice within the song the audio sample was matched	`String`	`getSongPosition`	Units are in milliseconds; only applies to MusicID-Stream
Adjusted Song Position	Indicates current position in song	`String`	`getAdjustedSongPosition`	Measures milliseconds since the beginning of the song. For the `idNow and recognizeFromMic` operations, this field returns the song position at the time `getAdjustedSongPosition` is called, assuming continuous song playback since the operation was initiated. Not supported for the `recognizeMIDStreamFromPCM` operation.
Track Mood	Track level mood descriptors	`ArrayList<GNDescriptor>`	`getMood`	
Track Number	One-based index of the track on the album	`int`	`getTrackNumber`	-1 if no trackNumber
Track Tempo	Track level tempo descriptors	`ArrayList<GNDescriptor>`	`getTempo`	
Track Title	Title of the track	`String`	`getTrackTitle`	
Track Title Yomi	Japanese phonetic representation of track title	`String`	`getTrackTitleYomi`	

GNAlbumIdSearchResult

Contains the result(s) of an albumId search operation.

Field	Description	Type	Accessor	Comments
No Match Responses	AlbumId results with No_Match	ArrayList<String>	noMatchResponses	
Error Responses	AlbumId results with GNAlbumIDFileError	ArrayList<GNAlbumIDFileError>	errorResponses	

GNAlbumIdSearchResponse

Describes the metadata fields for an album and the track on that album that was matched by the albumId search operation. Inherited from GNSearchResponse. Contains all GNSearchResponse objects and fileIndent used in albumId operation per file.

Field	Description	Type	Accessor	Comments
GNSearchResponse	All GNSearchResponse metadata			Inherited from GNSearchResponse

File Ident	File Identifier	String	fileIdent	

GNDescriptor

Defines a Gracenote descriptor. The genre, mood, tempo, origin, era, and artist type data delivered by Mobile Client is referred to as descriptors. This object describes a descriptor by providing its name and identifier.

Field	Description	Type	Accessor	Comments
Descriptor	Descriptor name	`String`	`getData`	
ID	Descriptor identifier	`String`	`getId`	

GNAlbumIdFileError

Container class for the errors that can occur in AlbumId operations.

Field	Description	Type	Accessor	Comments
Error Message	Indicates results with error	String	errMessage	
Error Code	Error code for file	String	errCode	
File Indent	File identifier	String	fileIdent	

GNLinkData

Holds a collection of Link identifiers and their providers.

Field	Description	Type	Accessor	Comments
Id	Actual value from the provider	`String`	`getId`	
Source	The provider of the Link identifier, such as "Amazon"	`String`	`getSource`	

GNCoverArt

Contains the cover art image for an album.

Field	Description	Type	Accessor	Comments
Data	Cover art image as a data stream	`byte[]`	`getData`	
MIME Type	Cover art data type. This can be used to read and render the cover art image appropriately.	`String`	`getMimeType`	
Size	The size of the cover art returned as THUMB, SMALL, MEDIUM, LARGE or XLARGE. For the size in pixels see Image Resizing Best Practice (see page 56).	`String`	`getSize`	

GNImage

Contains the contributor image for an album.

Field	Description	Type	Accessor	Comments
Data	Contributor image as a data stream.	`byte[]`	`getData`	
MIME Type	Contributor data type. This is used to read and render the contributor image appropriately.	`String`	`getMimeType`	
Size	The size of the contributor image returned as THUMB, SMALL, MEDIUM, LARGE or XLARGE. For the size in pixels see Image Resizing Best Practice (see page 56).	`String`	`getSize`	

Fingerprint Creation Result

A Mobile Client fingerprint creation result delivers the created fingerprint to your application in a simple hierarchy described below.

```
java.lang.Object
  :
  ..com.gracenote.mmid.MobileSDK.GNResult
      :
      .. com.gracenote.mmid.MobileSDK.GNFingerprintResult


          ..... Inherits from
```

GNFingerprintResult

Field	Description	Type	Accessor	Comments
Fingerprint Data	Fingerprint generated by operation. The fingerprint can be used to recognize the audio.	`String`	`getFingeprintData`	

Considerations

Best Practice for Receiving Full and Partial Metadata Responses

Set `GNConfig.allowFullResponse` to `true` only when your application requires full(extended) metadata.

If your application can live with partial metadata, always set `allowFullResponse` to `false`. Only ask for full metadata when required. This will improve your match/response time.

Best Practice for Receiving Results and Status Change Updates

The callbacks for handling status changes (`GNOperationStatusChanged` interface implementation) and results (`GNSearchResultReady` interface implementation) are called on the main thread. As a best practice, your app should keep the main thread readily available for these callbacks and run any other time-consuming tasks in the background.

Recognizing Audio from the Microphone

The following practices are recommended when recognizing audio from the microphone:

- Provide clear and concise onscreen instructions on how to record the audio:
    - Most failed recognitions are due to incorrect operation by the user.
    - Clear and concise instructions help the user correctly operate the application, resulting in a higher match rate and a better user experience.

- While recording audio from the microphone display a progress indicator:
    - When Mobile Client is recording from the microphone, the application can receive status updates. The status updates indicate what percentage of the recording is completed.
    - The status updates are issued at every tenth percentile of completion, meaning 10%, 20%, 30%, and so on.
    - Use this information to display a progress bar (indicator) to notify the user.
    - When recording has finished use vibration or a tone (or both) to notify the user.

- Visual only notifications can hamper the user experience because:
    - The user may not see the notification if they are holding the handset up to an audio source.
    - 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.
    - While Web Services is being contacted, display an animation that indicates the application is performing a function. If the application appears to halt the user may believe the application has crashed.
    - If no match is found, reiterate the usage instructions and ask the user to try again.

Recognizing Audio from a File using MusicID-File

The following practices are recommended when recognizing audio from a file when using MusicID-File. For best practices regarding AlbumID operations, please contact your Professional Services representative.

- Allow the user to select multiple files for recognition. Submit them concurrently for fastest results.
- Limit the number of stored results and pending recognitions:
    - File recognition operations can run concurrently, which allows many requests to be issued at once.
    - Each recognition operation and each returned result requires memory.
    - Be careful to limit the number of results stored and pending recognition operations, to avoid exhausting the device's RAM.

- While Web Services is being contacted, display an animation that indicates the application is performing a function. If the application appears to halt, the user may believe the application has crashed.

Image Resizing Best Practice

Summary

This topic provides guidance on image implementation by discussing Gracenote's use of predefined square dimensions to accommodate variances among image orientations.

Description

Gracenote provides a variety of images as part of our enhanced content offerings. For Mobile Client, these include Cover Art, Artist, and Genre images.

To accommodate variation in the images' dimensions, we resize the images to fit within the standardized image dimensions of a predefined-square, so that the longest dimension equals the dimensions of one side of the square, while the other dimension is somewhat short of the full square. The following images show how an image is resized within the predefined square. (Note that the outline for the square is used here only to demonstrate layout. We do not recommend displaying this outline on your application's user interface.)

Predefined Square
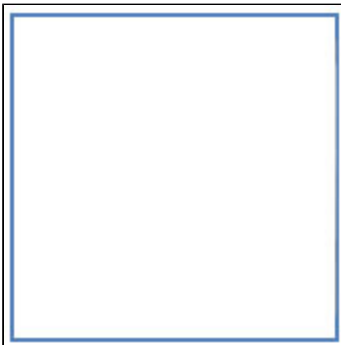
Horizontally-oriented Image Examples

Vertically-oriented Image Examples

The following sections discuss the differences between music images and give the standard image dimensions for each image type.

Music Cover Art
While CD cover art is often represented by a square, it is more accurately 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 a radically different shape.

Gracenote Standard Image Dimensions: Music Cover Art

Size	W x H (Pixels)
Thumbnail	75 x 75
Small	170 x 170
Medium	450 x 450
Large	720 x 720
Extra Large	1080 x 1080

Artist Images

Like music Cover Art, Artist images are seldom square, and are generally more obviously rectangular than music cover art. Because music artists range from solo performers to bands with many members, the images may either be wide or tall.

Gracenote Standard Image Dimensions: Artist Images

Size	W x H (Pixels)
Thumbnail	75 x 75
Small	170 x 170
Medium	450 x 450
Large	720 x 720
Extra Large	1080 x 1080

Image Implementation Recommendation

For all images: We recommend that the image be centered horizontally and vertically within the predefined square dimensions, and that the square be transparent so that the background shows through. This results in a consistent presentation despite variation in the image dimensions.

Searching by Text, Lyric Fragment, and Fingerprint

The following practices are recommended when searching by text, lyric fragment, and fingerprint.

- Allow user to input as much information as possible:
    - Text search allows artist name, album title, and track title to be entered. Allow the user to input all of these fields and pass them to Gracenote for the search.
    - Lyric fragment search can be augmented with the artist name. Allow the user to provide the artist name and ensure it is passed to Gracenote for the search.

- While Web Services is being contacted, display an animation that indicates the application is performing a function. If the application appears to halt the user may believe the application has crashed.

Retry Facility

Mobile phone handsets are sometimes unsuccessful in creating a network connection. This could be due to the mobile phone handset not being within range of a suitable network, or the network being temporarily unavailable due to device or network limitations.

It is recommended that you implement a retry facility that retries the user request several times before reporting to the user that the request could not be fulfilled. A retry facility allows temporary network unavailability to go unnoticed by the user.

Resource Management

Mobile Client is a compact library that provides easy access to rich content that can enhance the user's experience. Its compact size and simplicity are not indicative of the vast amount of content it can quickly deliver.

It is important that your application is prepared to correctly manage requests for content and is able to handle the content when it is delivered.

For more information on the following resource mangement topics, contact your Gracenote Professional Services representative.

Storing Content in RAM

The richness of the content delivered by Mobile Client and the speed of delivery can easily result in megabytes of data being transferred to the device. This can quickly consume the RAM resources available to applications, especially those operating in a resource-limited environment.

Applications must consider carefully how data is stored and how long it is stored.

If your application performs a single track recognition and then displays the result, the RAM resources are unlikely to be exhausted. But if the application performs thousands of track recognitions and intends to display all the returned data, then RAM usage will become an issue.

Applications should limit the amount of data they attempt to store in RAM. For example, an application that recognizes a user's entire song collection could limit the number of recognition results using paging. The application can recognize a small number of songs, display the results for the user to peruse, and then when prompted by the user, recognize the next set of songs. For more information, see Result Paging (see page 60).

Applications should also consider what needs to be stored in RAM. It might be acceptable for your application to store data returned from Mobile Client directly into persistent memory, only using RAM to display small subsets of the data to the user.

The schemes used to manage RAM resources will vary greatly with use case and implementation. It is important that dealing correctly with data delivered from Mobile Client be a major consideration for the development team early in the design phase.

Request Flow Control

Mobile Client is capable of processing multiple requests concurrently. This facility enables an application to obtain results faster than issuing one query at a time.

There is no limit to the number of requests that can be issued; Mobile Client queues the requests until they can be processed. Each request, whether it is queued or being processed, consumes RAM. If requests are issued faster than they can be processed, the application may run out of memory.

To avoid this situation an application should implement flow control for issuing requests. The number of requests pending (queued or being processed) should be capped at a maximum. It is recommended that you limit the number of pending requests to five (5).

Applications should issue requests until the number of pending requests reaches the predefined maximum. No further requests should be issued until the number of pending requests drops below the predefined maximum.

In addition, all layers in the application stack should implement request flow control, or be capable of rejecting requests from upper layers.

For example, if you are developing a middleware layer that interfaces upper layer applications with Mobile Client, your middleware layer should implement flow control. If the number of pending requests rises to the predefined maximum, the layer should reject any new requests until the number of pending requests drops below the predefined maximum. Sophisticated applications will also provide notifications to upper layers when requests can and can't be accepted (similar to a modem's XON and XOFF commands).

Result Paging

In addition to request flow control, an application with user navigable results should optimally use results paging to limit the number of requests issued and results stored. Paging can also be used to ensure the application is responsive.

Results paging can be effective in applications that can display a large number of results in sections, such as a scrolling display, or paged results where *Next* and *Previous* controls are used for result navigation.

It is recommended that such applications store the result for three (3) pages: the current page, the previous page, and the next page. As the user navigates through the pages, the results in the current, previous, and next pages change. As they change, the application can delete stored results and retrieve new results as they are needed to fill the current, previous, and next pages.

Using this scheme, the previous and next pages are pre-retrieved, making the application responsive as the user navigates.

Limiting the total number of results pre-retrieved also limits the amount of content that is stored in RAM. Gracenote recommends using five results per page. If the user is limited to moving one results page at a time, only one page of results needs to be retrieved with each move. This limits the number of concurrent queries to five, as recommended in .

Error Handling

When an operation cannot be completed, Mobile Client generates the appropriate error information which is returned via a result-ready Object. Your application should handle the error conditions and proceed accordingly.

In the event of an error, Mobile Client returns an error code and an error message. The error code can be used by your application to react to specific error conditions. The error message contains error condition information, which provides additional clues to the error's cause. Error messages are not suitable for display to the end-user, as they contain technical information and are English-language specific; they are also subject to change without warning. The error message displayed to the user should be derived from the error code.

Sophisticated implementations may provide an error notification mechanism that allows errors to be investigated in deployed applications. For example, your application can send error information to a support server or prompt the user to send error information to an email address. Once notification of an error has been received, technical personnel can investigate the error accordingly.

GNResult.FPXFailure and GNResult.FPXFingerprintingFailure

These errors are generated if Mobile Client could not generate a fingerprint from an audio sample. Your application should indicate that the audio cannot be recognized due to invalid or unsupported format.

GNResult.FPXListeningFailure

This error is generated when Mobile Client cannot obtain the microphone for recording an audio sample. Your application should indicate to the user that the microphone is required to record music for identification and to try again after closing any applications that may be using the microphone.

GNResult.WSRegistrationInvalidClientIdFailure

This error is generated when the Client ID provided to Mobile Client was not successfully authenticated by Web Services. Your application should indicate to the user that the music identification service is unavailable.

If you are notified of this error via an error notification mechanism notify your Gracenote Professional Services representative.

GNResult.WSNetworkFailure

This error is generated when the device does not have a valid network connection. This could be because the mobile phone handset is not within range of a suitable network or the network is temporarily unavailable due to device or network limitations.

Your application should implement a retry facility, retrying the query a number of times. If the problem persists after all retries, indicate to the user that a network connection could not be created and to try again later.

GNResult.WSFailure

This error is generated when Gracenote Web Services cannot fulfill a request from the Mobile Client or a valid network connection could not be made. The accompanying error message is a description of the error condition as provided by Gracenote Web Services.

Your application should indicate that the user's request could not be fulfilled at this time. Do not display the error message to the user as it contains technical language which is generally not user friendly. Also, do not attempt to parse the message as messages returned from Web Services change regularly without warning.

If your application provides an error notification mechanism, forward the error message to your technical personnel. If required, provide the message to your Gracenote Professional Services Representative for further analysis.

GNResult.WSInvalidDataFormatError

This error is generated when Mobile Client cannot parse the response received from Gracenote Web Services. This could occur due to corruption of the results data. Your application should indicate to the user that their request failed and offer to retry the query.

GNResult.UnhandledError

This error code is reported when an unexpected error occurs that is outside the normal operation of Mobile Client and outside the scope of Mobile Client's error capture capabilities. The possible causes of such an error are very broad and include but are not limited to platform errors, third-party library errors, hardware errors, etc.

Your application should indicate that an error occurred and the current request could not be completed.

Limiting Query Time

Mobile Client imposes time limits for the amount of time a device takes to create an Internet connection or fulfill an Internet request. The time taken to fulfill a request can depend upon:

- The device's internal resources (memory, MIPS, etc.)
- The capacity of the Internet connection (the amount of data that can be transported by the air interface)

- The availability of the Internet connection (devices can move in and out of signal range or the network may become congested)

The timeouts chosen by Mobile Client are based on the most limited device with the most limited Internet connection capacity and availability. For high end devices these timeouts may be considered too long.

It is recommended that devices impose their own query timeout mechanism that cancels a query when it exceeds an application specified time limit. This allows the application the flexibility to limit query lengths in accordance with the capabilities of the host devices.

When initiating a query to the Gracenote database via Mobile Client, start an application timer with an application-specified timeout. If the timer expires before the query completes, cancel the query using the `GNOperations.cancel()` method.

Applications can also reduce query times by limiting the amount of content that is downloaded in each query. It is often acceptable to perform a recognition query and receive only the basic metadata for the result and retrieve the related content in separate queries thereafter.

> ⚠ When calling AlbumID operations, do not set a timer. AlbumID directory operations can take a long time, depending on many factors, such as the number of tracks in the directory, processor speed, network speed, type, and other factors.

Improving Retrieval Performance by Using Enriched Content URLs

There may be cases when you wish to retrieve metadata, descriptors, or external identifiers as quickly as possible, while lazy-loading enriched content such as cover art in the background. To achieve this, Mobile Client features a set of APIs that allows you to extract the URL pointers to the enriched content, giving you greater control over when you load the additional content. Mobile Client provides the following APIs, each of which retrieves a different type of enriched content (cover art, artist images, album reviews and artist biographies):

- GNSearchResponse.getCoverArt().getUrl()
- GNSearchResponse.getContributorImage().getUrl()
- GNSearchResponse.getAlbumReviewUrl()
- GNSearchResponse.getArtistBiographyUrl()

Each API returns a string containing the URL for the enriched content. The returned value will be null if no enriched content is available. In the case of cover art, if there is no cover art available, the URL for genre art may be returned (provided the property for genre art is set).

> ⚠ Enriched content URLs are temporary; therefore, the application must be configured to handle expired URLs. Gracenote currently supports content URLs for a lifespan of one hour, but this may be subject to change. Please contact your Gracenote Professional Services representative to discuss best practices for temporary caching and error handling.

The following code examples show how to retrieve the enriched content URLs:

```
// Retrieve cover art URL
GNCoverArt coverArt = bestResponse.getCoverArt();
if(coverArt =! null) {
    String coverArtUrl = coverArt.getUrl();
}

// Retrieve artist image URL
GNImage artistImage = bestResponse.getContributorImage();
if ( artistImage != null ){
    String artistImageUrl = artistImage.getUrl();
}


// Retrieve artist biography URL
String artistBiographyUrl = response.getArtistBiographyUrl();

// Retrieve album review URL
String albumReviewUrl = response.getAlbumReviewUrl();
```

Error Handling

When the application accesses the URLs, it should handle HTTP errors gracefully. The following table lists the types of HTTP errors that might occur and the corresponding action to take:

Error	Action
HTTP 500 error	Retry fetching the URL once.
HTTP 412 error (Precondition failed)	The URL has expired. The application should request a new URL from the Mobile Client SDK, using `GNSearchResponse.getTrackId()` to get the Gracenote Identifier. The application should then use `GNOperations.fetchByTrackID()` before fetching the URL.
HTTP 4xx error (other 400-level errors)	Handle as if the URL has expired: request a new URL and try fetching the asset again. If that fails, the application should log an error and not attempt further retries

AlbumID Operations

As a best practice, the application should retrieve the URLs for tracks separately from `GNSearchResponse` after making an AlbumID request. The application should optimize and retrieve the cover art, artist image, album review, and artist biography URLs (and images and data) for one track in the album and not for each track.

Upgradability

When new features are added to your application or bugs are fixed, you may want to issue an upgrade to your users. An application's upgradability is a measure of how easily a new version can be distributed to users.

An application with high upgradability has the following features:

1. Automatically detects that a newer version is available
2. Prompts the user to upgrade to a newer version and gives the option to:
    1. Upgrade now
    2. Remind me later
    3. Never upgrade to this version
3. Provides the ability to force an upgrade (no option provided)
4. Does not overwrite any of the user's data when upgrading, including user history data
5. Has unique version numbers for all every application build released
6. Limits external dependencies (for example, referencing libraries not stored in the APK)

Application Versions

Upgradable applications take special care to ensure unique version numbers for every application build released. A process to ensure unique version numbers is essential to successfully implementing an upgradable application and infrastructure.

Android applications use a simple integer to denote the version where newer versions have a higher integer value. This allows the other applications, such as the upgrade infrastructure, to easily determine if one application version is newer than another.

Android also provides a version name that is a String. This is generally used as the formatted version displayed to users and can contain information such as major and minor release numbers.

The version code and version name can be defined in the application's manifest file.

Application Architecture

Applications with high upgradability minimize external dependencies. If a new version of your application requires a newer version of a library that is stored outside the APK, it cannot be successfully upgraded unless the external library is also upgraded.

Your application cannot initiate the upgrade of a library stored outside the APK.

This also applies to the Gracenote libraries. For your application to be upgradable, it must store the Gracenote libraries internally, within its APK. If the Gracenote libraries are stored externally to the APK, such as in the `/system/lib` directory, then a version of your application based on newer Gracenote libraries cannot be correctly installed.

Upgrade Methodology and Infrastructure

When developing the service that publishes your application, consider the upgrade methodology and infrastructure.

A sophisticated implementation provides an endpoint (URL or IP address) that the application can check periodically. The service provides the version number of the newest version available. The application can then determine if the newest version is newer than itself.

Similarly an option can be provided to allow the user to check for upgrades. When commanded by the user, the application retrieves the newest version number from the service.

When a newer version of the application is detected, a prompt can be provided to the user giving them the option to upgrade the application. The options provided to the user may be:

1. Upgrade now
2. Remind me later
3. Never upgrade to this version

Some customers will also build in the ability for the application to force an upgrade. Forced upgrades do not require user consent to upgrade to a new version. Forced upgrades are often used to:

1. Fix security issues in a previous version that threaten your service
2. Retire old technology that will no longer be supported

It is recommended that your application implement both optional and forced upgrade mechanisms.

Glossary

Android Activity Object	An Activity Object is one of the building blocks of an Android application. An Activity provides the application with context to render to the screen. An Android application typically has at least one Activity object.
Link Data	When query results are returned by Gracenote they can contain Link data which directly pertains to the result it is embedded in. A single piece of Link data consists of an identifier (Integer) and a source (String). The source is the provider of the identifier, such as Amazon.com. The identifier is a unique key into the providers catalog which can be used to present the user with additional content or services. A single result can contain multiple pieces of Link data from multiple providers.
Lyric Fragment	A segment of lyrics from a track.
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.
Result Ready Interface	Mobile Client defines Java interfaces that allow Mobile Client operations to deliver results to the application. These interfaces are known as Result Ready Interfaces.

Result Ready Object	An object that implements a Result Ready Interface is a Result Ready Object. An instance of a Result Ready Object must be provided when invoking a Mobile Client operation, so the operation can deliver results to the application.
Single Best Match	Certain audio recognition products provided by Mobile Client return a single result: the single best match. The single best match is determined by Web Services based on criteria specific to the application. This criteria can be refined by the application.

Troubleshooting

Sample Application Project Generates a Google API Error

Problem

The Mobile Client Sample Application project generates an error similar to the following:

> *Unable to resolve target 'Google Inc. Google APIs:4'*

Solution 1

The Gracenote Mobile Client Sample Application uses Google APIs to determine the user's location when identifying a song; consequently, the Google APIs must be included to correctly run the Sample Application project.

Use the **Android SDK and AVD Manager** functionality to download the appropriate version of the Google APIs into your Android development environment.

💡 More detailed instructions are available in the Gracenote technical note *Getting Started with the Android Sample Application* (included in this SDK).

Solution 2

If you already have a version of Google APIs in your Android development environment, this error could be generated if you have a version that is incompatible with the target of the Mobile Client Sample Application Android Project.

Enter the project properties (in Eclipse's Package Explorer, right click on the project name, and click on properties in the context menu that appears). In the project properties, select "Android". In the "Project Build Target" pane, select the build target to be the version of Google APIs you have installed. Do a clean build of the application. The application should now build.

Note:Google APIs are not required for Mobile Client; they are only required for the Mobile Client Sample Application.

Sample Application Project Generates a Class Compatibility Error

Problem

The Mobile Client Sample Application project generates an error similar to the following:

> *Android requires .class compatibility set to 5.0. Please file*

Solution

The Android project's code generation version is not compatible with the versions supported by the packages downloaded to your Android SDK. Change the class compatibility version to resolve this error.

To fix the class compatibility issue in Eclipse:

1. Right-click on *GN_Music_SDK* in the Package Explorer tab.
2. Navigate to the **Android Tools** menu option and select *Fix Project Properties* in the sub-menu.
3. Eclipse invokes the Android SDK to determine the correct class compatibility version and changes your Android project accordingly.

Sample Application Asks If Client Identifier Is Set

Problem

When starting the Sample Application, a message similar to the following displays:

> *Initialization error. Is Client ID set?*

Solution

Ensure your Client ID is correctly provided to GNConfig.init in method onCreate in GracenoteMusicID.java.

Invalid Client Identifier Exception

Problem

When calling GNConfig.init, an exception displays with the message:

> *invalid clientId*

Solution

Ensure your Client ID is correctly provided to GNConfig.init in the correct form, which is:

```
<Client ID>-<Client ID Tag>
```

Your Client ID and Client ID Tag are provided by your Gracenote Professional Services representative.

Can't Find Registration Information

Problem

Mobile Client returns the error message:

Can't find registration information

Solution

Check that you device has a valid network connection.

This error can occur if the operating system reports there is a network connection when the device is not connected to a network. This is generally caused by a delay between the device losing a network connection and the operating system reporting the connection was lost.

IO Exception While Connecting to Web Services

Problem

The following error occurs when performing a query:

[Mobile:5000] webservices http status: 500: IO exception while connecting to webservices

Solution 1

In the application's AndroidManifest.xml file, ensure the Internet permission is defined.
The following permission tag should be present:

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

For the complete set of permissions that must be defined, see the Sample Application's AndroidManifest.xml file.

Solution 2

Ensure your device and application have a working connection with the Internet.

This error can be seen due to loss of connectivity with the Internet or Internet traffic congestion.

Exception While Listening to Microphone

Problem

The following error occurs when performing a query:

> *[Mobile:3000] An exception occurred while recording from the microphone. The audio record state returned uninitialized.*

Solution

In the application's AndroidManifest.xml file, ensure the record from microphone permission is defined. The following permission tag should be present:

```
<uses-permission android:name="android.permission.RECORD_AUDIO" />
```

For the complete set of permissions that must be defined, see the Sample Applications AndroidManifest.xml file.

Network Failure: OpenSSLSocketImpl Unknown Error

Problem

Occasionally Android is unable to successfully create a connection due to a socket error. In the logCat, errors similar to the following will display:

```
E/OpenSSLSocketImpl( 2252): Unknown error 1
```

Or

```
E/OpenSSLSocketImpl( 2252): Unknown error 5
```

The Mobile Client will output a debug message similar to the following:

```
D/Gracenote MobileSDK(20284): IO exception while connecting to
webservicesjava.io.IOException: SSL handshake failure: I/O error during system
call, Unknown error: 0
```

When this condition arises Mobile Client generates a GNResult.WSFailure error.

Solution

The actual cause of the problem is unknown. As it has been seen on various devices, it is not device-specific. It is thought to be due to temporary unavailability of the network, perhaps due to platform or device limitations (or both), or the network itself.

There is no known way to prevent the error from occurring. However, as it is a temporary condition (usually lasting only a few seconds), It is recommended that you employ a retry mechanism to try the transaction again.

Unknown Host Exception

Problem

Android HTTP client generates a `webservicesjava.net.UnknownHostException`. This appears in the logCat output.

Solution

The exception is thrown by Android's HTTP Client because the Gracenote database servers cannot be reached. The exception is not an indication of a software error, rather it is a notification of why a HTTP request could not be fulfilled.

Nothing can be done to prevent the exception from being thrown if the Gracenote database servers cannot be reached, but Mobile Client does catch the exception and report a GNResult.WSNetworkFailure to your application.

The general cause of this exception is that the device is not within range of a wireless network, or a wireless network access is not enabled on the device. To prevent this exception, ensure that the application is connected to a network and has access to the Internet.

Advanced Implementations

Service-Based Implementations

These examples show utilizing the Mobile Client libraries in an Android Service. Use of a Service is an advanced concept requiring significantly more design and development than an Activity-based application. For this reason, activity-based implementations are recommended. However, for those cases when the Mobile Client functionality must run without a UI, we support running the Mobile Client in an Android Service.

Prerequisite: The included Mobile Client Sample application must be installed, complied, and functioning on a device before developing Android Service-based implementations.

To use the example code:

1. Copy and paste the code into your Eclipse project in different .java files.
2. Add a clientID string.
3. Register service with adding the following code in the AndroidManifest.xml file: `<service android:enabled="true" android:name="com.example.GracenoteService" />`
4. Add any other required code.
5. Compile and run.

For assistance with advanced implementations, contact your Gracenote Professional Services representative.

Service Control Application

The Service Control Application stops, starts, and provides the status of an Android Service.
This example code is not required to utilize Mobile Client functionality. It is recommended that you provide a friendly way for the end-user to know a background Service is running, to restart, and to stop. This functionality could be included in an application the Service interacts with.
Note: Since this application launches the Service, the application needs to be running for the Service to continue.

```java
/* Gracenote Android Music SDK Sample Service Launcher
 *
 * Copyright (C) 2010, 2011 Gracenote, Inc. All Rights Reserved.
 */
package com.example;

import com.example.GracenoteService;

import android.app.Activity;
import android.content.Intent;
import android.graphics.Color;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.TextView;
/**
 *    Example application that Stops, Starts, and provides the Status of an Android Service.
 *    This example code is NOT REQUIRED as this functionality can/should be built into an
 *    application the Service is supporting.  It is recommended that you provide a friendly
 *    way for the end-user to know a background Service is running, to restart, and to stop.
 */
public class ServicesDemo extends Activity implements OnClickListener {
  private static final String TAG = "GracenoteServiceLauncher";
  String startedStatusString = "Status: Started";
  String stoppedStatusString = "Status: Stopped";
  Button buttonStart, buttonStop;
  TextView textViewStatus;
  static boolean serviceStatus;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    buttonStart = (Button) findViewById(R.id.buttonStart);
    buttonStop = (Button) findViewById(R.id.buttonStop);

    textViewStatus = (TextView) findViewById(R.id.textStatus);
    if(serviceStatus){
        textViewStatus.setText(startedStatusString);
        textViewStatus.setTextColor(Color.GREEN);
    }else {
```

```
         textViewStatus.setText(stoppedStatusString);
         textViewStatus.setTextColor(Color.RED);
     }

   buttonStart.setOnClickListener(this);
   buttonStop.setOnClickListener(this);
 }

 public void onClick(View src) {
   switch (src.getId()) {
   case R.id.buttonStart:
     Log.d(TAG, "onClick: starting service");
     startService(new Intent(this, GracenoteService.class));
     textViewStatus.setText(startedStatusString);
     textViewStatus.setTextColor(Color.GREEN);
     serviceStatus = true;
     break;
   case R.id.buttonStop:
     Log.d(TAG, "onClick: stopping service");
     Boolean test = stopService(new Intent(this, GracenoteService.class));
     if (test) {
         Log.d(TAG, "service stopped");
         textViewStatus.setText(stoppedStatusString);
         textViewStatus.setTextColor(Color.RED);
         serviceStatus = false;
   }else {
       Log.d(TAG, "not able to stop service");
   }
     break;
   }
 }
}
```

Service Implementation

This use case shows recognizing music files located in a directory with MusicID-File.
A Client ID string is required and must be compiled into a java Service class. For details, see Configuration and Authentication (see page 10).

A Service implementation may save the results of a query (GNResult object) in a local database, file, or memory. This example shows saving the results to a file (response.log). For details on supported operations, see Operations (see page 11). Typical implementations are likely to loop, watching directories or receive messages from another application that invokes GNOperations. Event-based implementations and looping in the background as an Android Service requires careful design. For assistance with advanced implementations, contact your Gracenote Professional Services representative.

The following example code can be copied into an Eclipse project that has the Mobile Client SDK properly installed. It is recommended that the included Mobile Client Sample application be installed, compiled, and functioning on a device before developing Android Service-based implementations.

```
/* Gracenote Android Music SDK Sample Service
```

```
 *
 * Copyright (C) 2010, 2011 Gracenote, Inc. All Rights Reserved.
 */
package com.example;

import java.io.File;
import android.app.Service;
import android.content.Intent;
import android.os.Binder;
import android.os.IBinder;
import android.util.Log;

import com.gracenote.mmid.MobileSDK.*;
/**
*     Android "Service" that implements Gracenote Mobile Client interfaces for recognizing music files
(MID-File).
*     A ClientID string is required and must be compiled into a java Service class.
*     Code for interacting with a "bound" application likely desired for customer implementations. A
Service
*     implementation likely uses the result object (GNSearchResult) within this Service vs. logging it to a
*     file as shown for recognizing music files.  See "Implementation Guide" for details on supported
operations.
 */
public class GracenoteService extends Service implements GNSearchResultReady, GNOperationStatusChanged {
    private final Binder binder = new LocalBinder();
    private GNConfig config;
    // Default location for Gracenote file; change for customer specific location.
    String debugLog = "/sdcard/gracenote/response.log";

    /**
     * Class for clients to access. Because we know this service always runs in
     * the same process as its clients, we don't need to deal with IPC.
     */
    public class LocalBinder extends Binder {
        GracenoteService getService() {
            return (GracenoteService.this);
        }
    }

    /**
     * Return the communication channel to the service.
     */
    @Override
    public IBinder onBind(Intent intent) {
        return (binder);
    }

    /**
     * Called by the system when the service is first created.
* MUST BE IMPLEMENTED.
     */
    @Override
    public void onCreate() {
        // client ID string is REQUIRED (Gracenote provided): format MUST Be: NNNNNNNN-NNNNNNNNNNNNNNNNNNNN
        config = GNConfig.init("",this.getApplicationContext());
        File debugLogFile = new File(debugLog);
        if (debugLogFile.exists()) {
```

```
                        debugLogFile.delete();
            }

        }

        /**
         * Called by the system to notify a Service that it is no longer used and is
         * being removed.
         */
        @Override
        public void onDestroy() {

        }

        /**
         * Called by the system every time a client explicitly starts the service
         */
        @Override
        public void onStart(Intent intent, int startid) {
            recognizeFileMethod();
        }
// Define supported file formats that MID-File can decode into PCM.
        private boolean isSupportedFormat(String filename) {
            return (filename.endsWith(".wav") || filename.endsWith(".mp3")
                    || filename.endsWith(".mp4") || filename.endsWith(".m4a") || filename
                    .endsWith(".3gp"));
        }

        /**
         * Fingerprint files stored as files in /sdcard/gracenote/. For each file,
         * create a fingerprint and search for song metadata.
         */
        private void recognizeFileMethod() {

            File sdcard = new File("/sdcard/gracenote");
            if (sdcard.exists()) {
                String[] filenames = sdcard.list();

                for (String filename : filenames) {
                    if (isSupportedFormat(filename)) {
                        File file = new File(sdcard, filename);
                        String filePath = file.getAbsolutePath();
                        GNOperations.recognizeMIDFileFromFile(this, this.config,filePath);

                        // numFilesToFingerprint++;
                    }
                }
            } else {
                String resultMsg = "No supported music files were found on the sdCard.  Please install files in
the /sdcard/gracenote/ directory of your device or emulator.";
                writeToLog("Err: " + resultMsg);
                writeToLog("\n");
            }
        }

        /**
         * When the fingeprint/lookup operation is finished, this method will be
```

```
    * invoked to record results in a file. ACTUAL IMPLEMENTATION IS CUSTOMER SPECIFIC and not likely to be
    * a file.
    */
   public void GNResultReady(GNSearchResult result) {
       if (result.isFailure()) {
           // An error occurred so display the error to the user.
           String msg = String.format("[%d] %s", result.getErrCode(), result
                   .getErrMessage());
           writeToLog("Err: " + msg);
           writeToLog("\n");
       } else {
           if (result.isFingerprintSearchNoMatchStatus()) {
               // Handle case of lookup with no match
               writeToLog("No Match: ");
               writeToLog("\n");
           } else {
               GNSearchResponse bestResponse = result.getBestResponse();
               writeToLog("Album title: " + bestResponse.getAlbumTitle());
               writeToLog("Artist name: " + bestResponse.getArtist());
               writeToLog("Track title: " + bestResponse.getTrackTitle());
               writeToLog("Track Genre: " + bestResponse.getGenre());
               writeToLog("\n");
           }
       }
   }

   /**
    * Intermediate status update from the fingerprinter. Could be used to status the bound application.
    */
   public void GNStatusChanged(GNStatus status) {

   }

   /**
    * Writes results of a query to log. Customer implementations expected to differ.
    */
   private void writeToLog(String msg) {
       File debugLogFile = new File(debugLog);
       // Commented out so result output log file is not deleted---for testing only.
       // if (debugLogFile.exists()) {
       // debugLogFile.delete();
       // }
       try {
           File parentDir = debugLogFile.getParentFile();
           if (!parentDir.exists()) {
               parentDir.mkdir();
           }
           if (msg.charAt(msg.length() - 1) != '\n') {
               StringBuilder builder = new StringBuilder(msg.length() + 1);
               builder.append(msg);
               builder.append('\n');
               msg = builder.toString();
           }
           if (parentDir.exists()) {

               // If parent dir does not exist after mkdir(), likely
               // that /sdcard does not exist.
```

```
                boolean worked = GNUtil.appendUTF8(debugLog, msg);
                if (!worked)
                    Log.d("SAMPLE_APP", "Can't write to log file.");
            }
        } catch (Exception e) {
            Log.d("SAMPLE_APP", "Can't write to log file.");
        }
    }
}
```

Considerations

Interacting with an Application

Most use cases are best addressed by creating a Service that interacts with *only one application*. Doing this eliminates complexity with interprocess communication.
If interacting with multiple applications, Service code will be required to ensure the responses from the Mobile Client are delivered to the appropriate application. *Using the Mobile Client with multiple applications may require each application to be validated by Gracenote*. Discuss your planned implementations with Gracenote Professional Services for advice.

Interruptions

When running an Android Service in the background, the Service must handle situations like a phone call interrupting a Mobile Client operation. Although many of these events are automatically handled when using an "activity" based application, background Service implementations must handle these events.

Background Operation

If the Service continually runs in the background, code is required to handle receiving events, looping, and restarting. For example, if a Service is designed to monitor directories for new music placed on the device, code is required to check the directories on some interval, compare the new files with a cache/database of prior recognitions, call the appropriate Mobile Client Operation, save the results, and then start the cycle again.

Installation and Starting

For a Service to be installed and started on a device, you must develop a method that includes the Service as a part of an application (.apk) or registered as a Service or some other method. The simplest approach that works on all versions of Android is to have an installed widget-based application that registers the Service as well as checks its status. The alternative is to have a primary application include the code to start the Service.

Error Handling

Handling errors with a background Service that has no UI requires careful design. A simple approach is to pass status messages to the calling application. Contact your Gracenote Professional Services representative for advice.
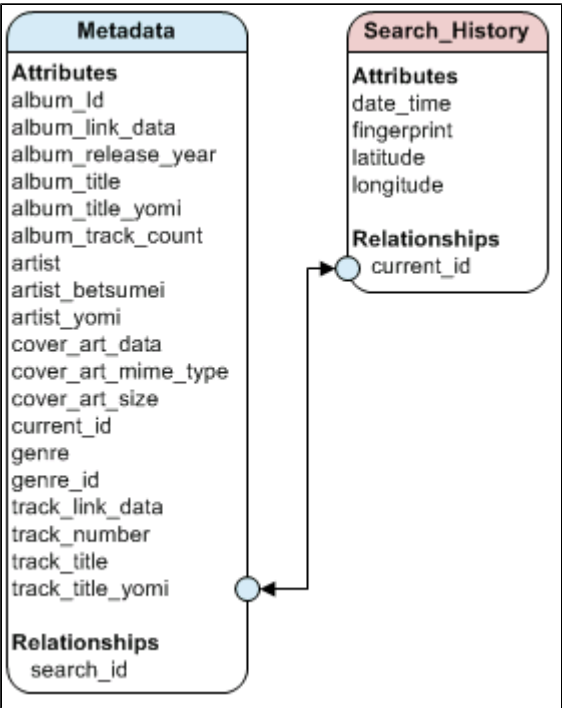
User History

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

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

Data Model

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

Mobile Client Sample Application User History Database Data Model

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

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

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

Adding Entries

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

- Recognizing music from the microphone
- Recognizing music from a PCM sample

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

Limiting Database Size

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

Recalling Entries

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

UI Best Practices Using MusicID-Stream

Gracenote periodically conducts analysis on its MusicID-Stream product to evaluate its usage and determine if there are ways we can make it even better. Part of this analysis is determining why some MusicID-Stream recognition queries do not find a match.

Consistently Gracenote finds that the majority of failing queries contain an audio sample of silence, talking, humming, singing, whistling or live music. These queries fail because the Gracenote MusicID-Stream service can only match commercially released music.

Such queries are shown to usually originate from applications that do not provide good end user instructions on how to correctly use the MusicID-Stream service. Therefore Gracenote recommends application developers consider incorporating end user instructions into their applications from the beginning of the design phase. This section describes the Gracenote recommendations for instructing end users on how to use the MusicID-Stream service in order to maximize recognition rates and have a more satisfied user base.

This section is specifically targeted to applications running on a user's cellular handset, tablet computer, or similar portable device, although end user instructions should be considered for all applications using MusicID-Stream. Not all recommendations listed here are feasible for every application. Consider them options for improving your application and the experience of your end users.

Clear and Accessible

All instructions provided to the user should be easy to understand and accessible at any time. 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.

Rotating Help Message upon Failed Recognition

When a recognition attempt fails, display a help message with a hint or tip on how to best use the MusicID-Stream service; a concise, useful tip can persuade a user to try again. Have a selection of help messages available; show one per failed recognition attempt, but rotate which message is displayed.

Allow the User to Provide Feedback

When a recognition attempt fails, allow the user to submit a hint with information about what they are looking for. Based on the response, the application could return a targeted help message about the correct use of MusicID-Stream.

Audio Listening Animation

While the device is recording an audio sample, display a simple image or animation that explains how to correctly use MusicID-Stream.

Audio Listening Progress Indicator or Countdown

Use a progress bar or countdown to indicate how long the application will be recording. The user can use this information to assist in collecting an audio sample that can be successfully recognized.

Audio Listening Completed Cues

Some failing MusicID-Stream recognitions are caused by insufficient or incomplete recording of the song. To assist the user in assessing if recording is complete, visual, audible and tangible cues can be used, such as:

- Display a message (only useful if the user is looking at the display)
- Sound a tone or tune (not useful in load environments)
- Vibrate the handset (always useful)

Use Street Sign-like Images

Street sign images are universal and easily recognizable. These can be used to provide clear instructions about where and how to use and not to use the MusicID-Stream service.

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.

Legal Notices

The following third party software is distributed with the Gracenote Mobile Client 2.5.2 (and later) software:

MPEG4IP

The source code version of this third party software is subject to the Mozilla Public License Version 1.1 (the "License"); you may not use libgnmc_aactag.so except in compliance with the License. You may obtain a copy of the License at http://www.mozilla.org/MPL/

Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License.

The Initial Developer of the Original Code is Cisco Systems Inc.
Portions of this code have been developed by Gracenote, Inc. Such contributions are Copyright 2005-2011 Gracenote, Inc. All Rights Reserved.

Contributor(s): Gracenote, Inc.

Base64Coder

Copyright 2009, Sebastian Stenzel, sebastianstenzel.de. All rights reserved.

This software is derived from the Base64Coder by Christian d'Heureuse.
Copyright 2003-2010 Christian d'Heureuse, Inventec Informatik AG, Zurich, Switzerland. All rights reserved.
www.source-code.biz (http://www.source-code.biz), www.inventec.ch/chdh (http://www.inventec.ch/chdh)

Licensed herein under BSD-2-Clause License, which can be found at
http://www.opensource.org/licenses/bsd-license.php (http://www.opensource.org/licenses/bsd-license.php)

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Packet Video

This software contains code for mp3 and AAC decoding derived from Packet Video (Copyright 1998-2009, Packet Video) and includes modifications developed by Gracenote, Inc. Such modifications are Copyright 2010-2012 Gracenote, Inc.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0. (http://www.apache.org/licenses/LICENSE-2.0)

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Protobuf

This software contains code for Protobuf.

Copyright 2000, Dave Benson.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Android Opencore source

This software contains code for decoding audio files and includes modifications developed by Gracenote, Inc. Such modifications are Copyright 2011 Gracenote, Inc.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Android Opencore source

Copyright (C) 2008, Google Inc. This software contains code for scanning media files for metadata including various modifications by Gracenote, Inc. Such modifications are Copyright 2010 Gracenote, Inc.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

MD5 Message-Digest Algorithm

Copyright (C) 1990, RSA Data Security, Inc. All rights reserved. License to copy and use this software is granted provided that it is identified as the "RSA Data Security, Inc. MD5 Message-Digest Algorithm" in all material mentioning or referencing this software or this function.

License is also granted to make and use derivative works provided that such works are identified as "derived from the RSA Data Security, Inc. MD5 Message-Digest Algorithm" in all material mentioning or referencing the derived work.RSA Data Security, Inc. makes no representations concerning either the merchantability of this software or the suitability of this software for any particular purpose. It is provided "as is" without express or implied warranty of any kind. These notices must be retained in any copies of any part of this documentation and/or software.