Cast Companion Library for Android

Last updated for v2.7.3

- Introduction
- Overall Design
  - Dependencies and Project Setup
- How to Use the Cast Companion Library
  - Initializing VideoCastManager
  - Adding Cast Button to Action Bar/Toolbar
  - Adding MediaRouteButton
  - Introducing Cast to new users
  - Adding Mini-Controller
  - Notifications
  - Lock Screen Controllers and Volume
  - Cast Player Screen
    - Volume Control
  - Session Recovery (Reconnection)
  - Handling Tracks and Closed Captions
    - API Support.
    - Changing the Style for Text Tracks
    - Setting Active Tracks from UI
  - Queues
  - Single Custom Data Channel
- Support for data-centric applications
- Hooking into Lifecycle Callbacks
- Advanced Topics
  - Custom Notification
  - Live Streams
    - Reconnection
  - Obtaining Authorization prior to playback
  - Supporting Configuration Changes in VideoCastControllerActivity
- Configuration and Setup
  - Manifest file
  - Configurable Messages
  - Setting up CCL for your application

Introduction

This document describes the Cast Companion Library for Android (CCL). Throughout the document, there may be references to the Cast Video application that is built using the CCL to show how the library can be used in a practical example.

CCL is written with the following objectives in mind:

- To provide a wrapper around the Cast SDK and related Google Play Services to hide the mundane tasks that are needed for casting.
- To provide a collection of ready-to-use Cast-related components and features that are strongly recommended by the UX Guidelines.
- To provide an example of how the Cast SDK can be used to accomplish more complex tasks.

Since playing media is a common use case on a Chromecast device, CCL provides extensive support for casting and managing media content. In addition, it supports applications that are strictly using custom data channels.

Here is a collection of features that CCL provides for media centric applications:

- Wrapper for device discovery
- Customized notification
- Customized Cast Menu (Media Router controller dialog)
- Lock Screen remote control via `MediaSessionCompat` and `NotificationCompat.MediaStyle`
- Player Control while casting
- Global access via mini-controller
- Pain-free session recovery, including the reconnection logic outlined in the UX guideline
- A single custom data channel for sending and receiving custom data
- Ability to check the installed Google Play services for compatibility
- Handling media tracks, including settings page for all versions of Android
- Handling changes to the volume of the cast device using the hard volume button inside and outside of the app, even on lock screen or when the screen is off
- Providing an overlay view to introduce Cast feature to new users

And for data centric applications:

- Wrapper for device discovery
- Wrappers for registering and using multiple custom data channels
- Pain-free session recovery
- Ability to check the installed Google Play services for compatibility
- Providing an overlay view to introduce Cast feature to new users

In the subsequent sections, we will describe how each of these features can be accessed by an application. In what follows, we use “client” to refer to the application that uses CCL.

Overall Design

Most of this document focuses on media-centric applications. Toward the end, we come back and discuss how data-centric apps can use this library.

To maintain the state of the application in regards to connectivity to a Cast device, there needs to be a central entity that transcends the lifecycles of individual activities in the application. The CCL's main component, the `VideoCastManager` is the entity that does that[1]. This class is a singleton that is responsible for managing the majority of the work by providing a rich set of APIs and callbacks, maintaining the state of the system (such as connectivity to the Cast device, status of the media on the receiver, etc), updating various components such as the CCL Notification Service and Mini-Controller.

In order to have a notification that lasts even if the client application is killed (directly by user or indirectly by the system) while casting, CCL starts a Cast Notification Service. The Cast UX guidelines request that the system notification for a Cast application to be visible only when the application is not visible. By providing simple hooks for the client, CCL handles this task as well.

Here is a subset of classes and their main responsibilities that will be discussed in more details in subsequent sections:

- *`BaseCastManager`*. An abstract class that handles most of the connectivity to a cast device, including the reconnection logic. Clients are encouraged to use the two concrete subclasses (*`VideoCastManager`* or *`DataCastManager`*) but if they need to modify the behavior, they can extend these two concrete classes or implement this abstract class directly.
- *`VideoCastManager`*. Specifically designed for video-centric applications, this is a singleton that clients interact with directly. It internally uses other classes and components of CCL to manage and maintain the state of a video-centric application.
- *`VideoCastNotificationService`*. A local service that allows notifications to appear when needed beyond the availability of the main application.
- *`MiniController`*. A compound control that provides a mini-controller for the client.
- *`VideoCastControllerActivity`*. Provides a default implementation of the Player Control screen that clients can use with their application. Most of the associated logic is defined in the *`VideoCastControllerFragment`*.
- *`DataCastManager`*. Another concrete subclass of the BaseCastManager that is specifically designed for data-centric applications. It is a singleton that clients interact with directly and internally uses other classes and components of CCL to manage and maintain the state of the data-centric application and provides means to register one or more namespaces for data communication between a sender client and a receiver application.

[1] `DataCastManager` is the corresponding one for the data centric use cases.

- *ReconnectionService*. A background service that handles reconnection logic when wifi connectivity is lost.

The MediaRouter and Cast SDK, in collaboration with the Google Play services, provide a collection of asynchronous calls that start with the discovery method and continues to route selection, device connection, `RemoteMediaPlayer` creation, etc. For each of these asynchronous calls, there is one or more callbacks that inform the caller of the success or failure of the API calls, and are generally a signal that moving to the next call is now permitted. Not all the activities or components in a client app are interested in receiving all the callbacks. CCL makes it easy for client components to only register for a subset of callbacks based on their needs. Although most of the Cast API calls are made by CCL, clients can still be notified of all relevant events and callbacks, if they choose to.

Dependencies and Project Setup

Here is a list of dependencies for CCL:

- android-support-v7-appcompat
- android-support-v7-mediarouter[2] (this has a dependency on android-support-v7-appcompat)
- Google Play Services - the Base and the Cast components.

Your client project then needs to list CCL as its only dependency for Cast related functionalities. In other words, your client application does not need to directly import or declare a dependency on the support libraries that are mentioned above since CCL brings those dependencies along[3].

Since Cast APIs are mainly provided by the Google Play Services, the CCL library provides a convenient static method that all client applications should call at their startup to verify that a compatible version of the Google Play Services is available on the device. If the Google Play Services is missing or needs to be updated, a dialog will inform user and direct her to go to the Play Store to download/update the appropriate version. If, however, the Google Play Services is installed but is not enabled, it will direct the user to the device settings page to address the issue. To enable this important validation check, here is a sample code that could calls this functionality in the onCreate() of the launcher Activity of your application:

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
```

[2] Be careful that you need to use v7 support version of MediaRouter library and not the one that is included in the Android framework

[3] If you use the AAR packaged version of this library in your project, then you would need to bring in these dependencies to your own project explicitly but if you import this library as a module, you shouldn't need to.

```
        BaseCastManager.checkGooglePlayServices(this);
        .....
    }
```

How to Use the Cast Companion Library

In what follows, we will mainly focus on the video-centric applications and use the `VideoCastManager` but in many cases, the same applies to the `DataCastManager` as well.

Initializing `VideoCastManager`

The client application needs to instantiate and initialize an instance of `VideoCastManager` in the `onCreate()` of the Application's instance. A number of features in CCL can be enabled or configured to match the needs of its client; these configuration parameters are captured in the `CastConfiguration` class. Clients need to build and configure an instance of this class prior to initialization of the library. To build this class, use the Builder pattern and set all the configuration parameters that are needed (we will explain these parameters later in this document); here is a sample snippet:

```
CastConfiguration options = new CastConfiguration.Builder(applicationId)
        .enableAutoReconnect()
        .enableCaptionManagement()
        .enableDebug()
        .enableLockScreen()
        .enableWifiReconnection()
        .enableNotification()
        .addNotificationAction(CastConfiguration.NOTIFICATION_ACTION_PLAY_PAUSE, true)
        .addNotificationAction(CastConfiguration.NOTIFICATION_ACTION_DISCONNECT, true)
        .build();
```

The above configuration object will instruct CCL to enable a number of features (Auto Reconnection, Caption Management, Debugging, Lock Screen controllers, customized MediaRouteDialogFactory, WiFi reconnection and Notification service. In addition, it tells CCL what actions should be available in the notification. Note that more features and parameters can be configured here; consult the JavaDocs for the `CastConfiguration` class to see a complete list.

Once a configuration object is built, it is easy to initialize the library[4] in your Application's instance::

[4] `initialize` will log an error message if it detects that an incompatible version of Google Play Services is installed, is missing or is not activated.

```
public void onCreate() {
    super.onCreate();
    CastConfiguration options = … // see above
    VideoCastManager.initialize(this, options);
}
```

After this initialization, one can access this singleton instance by calling:

```
mCastManager = VideoCastManager.getInstance();
```

***Remark**. In performing certain tasks, CCL needs to have a reference to the Application `Context` instance from the client application (for example, to access client resources or system services). Any action that requires an Activity Context will ask for one from the client in the respective api call.*

Adding Cast Button to Action Bar/Toolbar

After having `VideoCastManager` set up, the next step is to put the Cast button in the Action Bar or Toolbar. The assumption here is that your activity is a direct or indirect subclass of `AppCompatActivity`, from appcompat-v7 library (see the next section for adding `MediaRouteButton` when your `Activity` inherits from `FragmentActivity`):

A. Add the following snippet to the xml file that defines your menu:

```
<item
     android:id="@+id/media_route_menu_item"
     android:title="@string/media_route_menu_title"
     app:actionProviderClass="android.support.v7.app.MediaRouteActionProvider"
 app:showAsAction="always"/>
```

B. Add the following line to the "`onCreateOptionMenu`()" of all your activities:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    super.onCreateOptionsMenu(menu);
    getMenuInflater().inflate(R.menu.main, menu);
    mCastManager.addMediaRouterButton(menu, R.id.media_route_menu_item);
    return true;
}
```

Note that the method `addMediaRouterButton` returns a pointer to the `MenuItem` that represents the Cast button, if you need to have access to it.

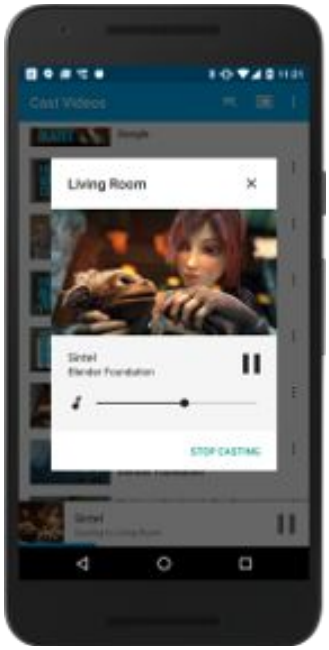
C. In `onResume()` of your activity, make sure to add the following snippet:

```
mCastManager = VideoCastManager.getInstance();
mCastManager.incrementUiCounter();
```

and in `onPause()`, call:

```
mCastManager.decrementUiCounter();
```

Now you have a Cast button in your Action Bar or Toolbar and the `VideoCastManager` will be in charge of all the required plumbing to provide the appropriate dialogs, etc.. CCL also provides a custom Cast MediaRouter Controller dialog when the client is connected:



Adding `MediaRouteButton`

If your `Activity` inherits (directly or indirectly) from the `FragmentActivity`, you have an additional option of using the `MediaRouteButton`. To add that button, add a snippet like the following to your layout:

```
<android.support.v7.app.MediaRouteButton
     android:id="@+id/media_route_button"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:mediaRouteTypes="user"
     android:visibility="gone" />
```

Then obtain a reference to this component in your activity and use the CCL library to set it up:

```
mMediaRouteButton = (MediaRouteButton) findViewById(R.id.media_route_button);
mCastManager.addMediaRouterButton(mMediaRouteButton);
```

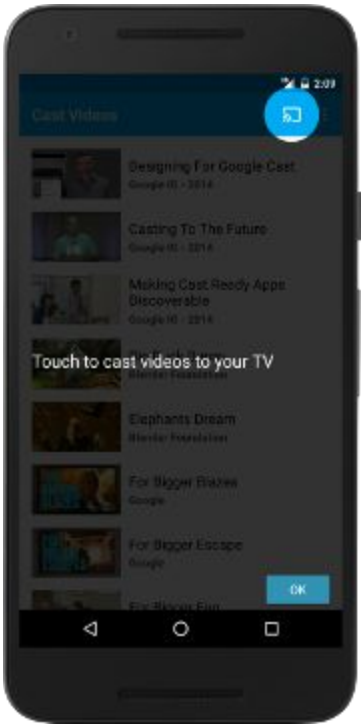
For updating the visibility, however, you need to hook into a callback from the `VideoCastManager`:

```
mVideoCastConsumer = new VideoCastConsumerImpl() {
    .....
    @Override
    public void onCastAvailabilityChanged(boolean castPresent) {
        mMediaRouteButton.setVisibility(castPresent ? View.VISIBLE : View.INVISIBLE);
    }
    .....
}
```

Make sure that similar to the previous section, you call incrementUiCounter() and decrementUiCounter() in your onResume() and in onPause() of your activity, respectively. In addition, you would need to obtain and set the initial state of the button when you enter an activity (the above callback will help you maintain the correct state when there is a change but the initial state needs to be set first). To that end, use the following snippet:

```
mMediaRouteButton.setVisibility(mVideoCasrManager.isRouteAvailable()
         ? View.VISIBLE : View.INVISIBLE);
```

Introducing Cast to new users



To make sure your users are aware that your application supports casting, it is recommended to highlight the cast button when the user opens your app for the first time.

To this end, CCL provides a simple overlay view that can be easily shown when the cast button appears in your application for the first time. To accomplish this, configure the callback `onCastAvailabilityChanged()` in your main activity to be notified when the cast icon becomes visible. Then configure an instance of `IntroductoryOverlay` to highlight the cast button:

```
mCastConsumer = new VideoCastConsumerImpl() {

    @Override
    public void onCastAvailabilityChanged(boolean castPresent) {
        if (castPresent && isHoneyCombOrAbove) {

            new Handler().postDelayed(new Runnable() {

                @Override
                public void run() {
                    if (mMediaRouteMenuItem.isVisible()) {
                        showOverlay();
                    }
                }
            }, 1000);
        }
    }
};

private void showOverlay() {
    IntroductoryOverlay overlay = new IntroductoryOverlay.Builder(this)
            .setMenuItem(mMediaRouteMenuItem)
            .setTitleText(R.string.intro_overlay_text)
            .setSingleTime()
            .build();
    overlay.show();
}
```

Here, `mMediaRouteMenuItem` is the reference returned by the `addMediaRouterButton()` that was used above and points to the `MenuItem` associated with the Cast button. If you are using a `MediaRouteButton`, you can point to hat by calling `Builder.setMediaRouteButton()`.
The UI for this overlay is fully customizable:

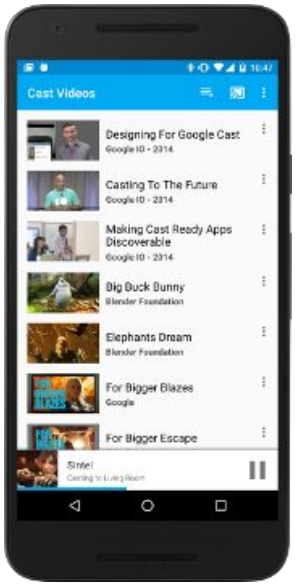
1. If you want to change the default layout, copy `res/layout/ccl_intro_overlay.xml` to your own project and update the layout while keeping the component ids the same.
2. To change the styling, take a look at `res/values/intro_overlay.xml` and copy the styles from there to your project and update them as needed.
3. The following resources can be directly updated in your project:

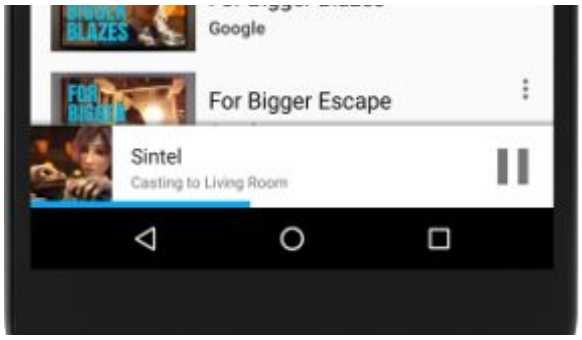
Resource Name	Type
`ccl_intro_overlay_background_color`	Color
`ccl_intro_overlay_button_background_color`	Color
`ccl_intro_overlay_button_foreground_color`	Color
`ccl_intro_overlay_button_text`	Text
`ccl_intro_overlay_focus_radius`	Dimension
`ccl_intro_text_margin`	Dimension
`ccl_intro_button_margin_right`	Dimension
`ccl_intro_button_margin_bottom`	Dimension
`ccl_intro_overlay_focus_radius`	Dimension

Adding Mini-Controller

The mini-Controller is a small persistent component that enables users to quickly see the content that is playing on the Cast device, to perform basic operations on the remote media (such as play/pause) and to provide a way to get back to the Cast Player page (by clicking on album art). It also provides an extended component to show the “upcoming” item in your queue, if you have a queue with auto-play turned on. The mini-Controller should be visible if and only if the remote video is playing or paused[5].

[5] For Audio-only content such as music, it may be required to have this component available even for local playback.





The CCL provides a compound component for the mini-Controller that can be added to the layout XML of your pages. Here is an example:

```
<com.google.android.libraries.cast.companionlibrary.widgets.MiniController
    android:id="@+id/miniController"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
    android:background="@drawable/shadow7"
    app:auto_setup=”true”
    android:visibility="gone">
</com.google.android.libraries.cast.companionlibrary.widgets.MiniController>
```

To configure this component, there are two choices:

1. (Preferred) Using the xml attribute `auto_setup = “true”`. If you set this xml attribute to `true`, then no additional work is required since the library does all the setup and required plumbing. This is the preferred approach.
2. Manual configuration. If you set `auto_setup` to false (this is the default value for this attribute), then the client application needs to register this component so that the library can manage its visibility, state and content. Additionally, the client application is responsible for unregistering this component when is not needed. To register this component with the instance of `VideoCastManager`, add the following lines to your Activities’ “`onCreate()`”:

```
mMiniController = (MiniController) findViewById(R.id.miniController);
mCastManager.addMiniController(mMiniController);
```

Remember to unregister this component, when you are done, by adding the following line of code to "`onDestroy()`" for example:

```
mCastManager.removeMiniController(mMiniController);
```

Whether you use the "`auto_setup`" feature, or manually register this component, the `VideoCastManager` handles updating the metadata shown in this component, the status of the play/pause button, and also the visibility of the whole control (e.g. if you disconnect your client application from the Cast device, it will hide this component). Note that the play/pause/stop buttons are drawables that are defined as aliases to the real assets so you can easily customize them in your application if needed. These aliases can be found in the res/drawable directory of the CCL and are named as i`c_mini_controller_pause` and similar names for the other two. You can copy those aliases to your application and change what they should point to.

When you are playing a queue of items and auto-play is on, this component shows an extended area when an upcoming item starts preloading on the receiver. This extended area is used to both inform the user of the upcoming item and also to let user control the auto-play behavior by providing a "play" button (if clicked on, the upcoming item will start playing immediately) or a "stop" button, to stop the app from proceeding to the next item; in this case, the playback of the current item will not be affected.

Notifications

Notifications will be provided for you if you enabled that feature in `CastConfiguration` instance that was used to initialize the `VideoCastManager`. What remains to be done is to help CCL discover when the client application is visible (to hide the notification) and when it is in the background (to show the notification). This will be accomplished by the following line that you have already added to your Activitys' "`onResume()`":

```
    mCastManager.incrementUiCounter();
```

and the following line that was added to "`onPause()`" of all your activities:

```
    mCastManager.decrementUiCounter();
```

Recall that these lines were already added for managing the Cast button so your activities should already have these. CCL also handles notifications on different versions of Android and uses the `NotificationCompat.MediaStyle` internally. Notifications, according to the Cast UX Checklist, should be only visible when your app is not in front. This will be handled by the library so no additional work is required.

Having a notification when the app is not in front provides a quick access point to the content that is being casted; user can easily open the notification and interact with the cast device. As such, it is required to provide a minimum set of controls on the notification. The required minimum is to include a button to play/pause the content, a button to disconnect from the cast device quickly, and a way to bring up the Cast Player screen. Depending on the nature of the app, clients may decide to include more actions. To this end, CCL provides 6 pre-configured actions that a client application can choose from for its notification. These are: Play/Pause, Play Next, Play Previous, Fast Forward, Rewind, and Disconnect from Cast. In addition, when adding these notifications, clients can decide which ones should be visible in the compact view of the notification. The configuration for this feature is done through the `CastConfiguration` object:

```
CastConfiguration options = new CastConfiguration.Builder(applicationId)
        .enableAutoReconnect()
        … /* some lines are not shown here */
        .enableNotification()
        .addNotificationAction(CastConfiguration.NOTIFICATION_ACTION_SKIP_PREVIOUS, false)
        .addNotificationAction(CastConfiguration.NOTIFICATION_ACTION_SKIP_NEXT, false)
        .addNotificationAction(CastConfiguration.NOTIFICATION_ACTION_PLAY_PAUSE, true)
        .addNotificationAction(CastConfiguration.NOTIFICATION_ACTION_FORWARD, false)
        .addNotificationAction(CastConfiguration.NOTIFICATION_ACTION_DISCONNECT, true)
        .setForwardStep(10)
        .build();
```

The red lines above show how you can first enable the notification feature (`enableNotification()`) and then configure what actions should be added by calling `addNotificationAction(int id, boolean showInCompact)`. The "`id`" here refers to the constants defined in the same class for the predefined actions mentioned earlier. The boolean flag `showInCompact` controls whether that action should be visible in the compact view of the notification or not. Note that there can be at most 5 actions, and a maximum of three can be visible in the compact view. In addition, for the Fast Forward and Rewind actions, the call `setForwardStep(int duration)` can override the default value, 30 seconds, used there[6]. Finally, it is possible for a client application to register a completely different Service to be used for handling notification; see the Custom Notification below for more details.

Side Note. You may wonder why we need to add those two lines (`incrementUiCounter()` and `decrementUiCounter()`) to all your activities. The main reason behind this is the following: CCL needs to know when your application is in front and when it is in background; this is important for multiple reasons; for example, when your app is in the background, we stop scanning for cast devices on the local network (to preserve battery and resources), or to hide the notification when your app is in front. In order to do this, CCL needs to stay up-to-date with the lifecycle of

[6] CCL provides two specific icons for Fast Forward and Rewind actions when the duration is set to 10 or 30 seconds. For any other values of duration, a generic icon will be used. All these icons are defined throw aliases so they can be easily changed by the client application.

your activities. In API 14+, one can obtain this information by using `Application.ActivityLifecycleCallbacks` but such API does not exist in earlier versions of Android and since this library supports API versions from 10+, a different mechanism was needed to satisfy this need. By using those two lines and hooking them to the lifecycle of all activities, CCL tracks when no activity is in front or otherwise.

Lock Screen Controllers and Volume

CCL can provide default lock screen controllers based on the combination of `NotificationCompat.MediaStyle` and `MediaSessionCompat`. If desired, this feature needs to be enabled when building the instance of `CastConfiguration` used in the initialization of the `VideoCastManager`.

In addition, when this feature is enabled, CCL provides the ability for users to control the (system) volume of the Cast device even if the application is in the background. On KitKat+ devices, it provides that capability even if the screen is off.

Cast Player Screen

If during the build of `CastConfiguration` you opted for the default Cast Player screen (that is the default behavior), you will get that without any additional effort.

The Cast Player page shows a full screen artwork with some additional metadata and some controls to manage the playback of the media on a remote device. It is important to provide a larger size image for your media for the best experience.

If user clicks on the artworks in the mini-controller or notification, a transition to the Cast Player page will happen automatically. If, however, you want to start casting from within the client application, the Cast Manager provides a simple mechanism for you to launch that screen:

```
mCastManager.startCastControllerActivity(context, mSelectedMedia, position,
autoPlay);
```

Here, `mSelectedMedia` is the `MediaInfo` object that represents your desired media that needs to be casted, (`int`) `position` is the starting point of the video and the (`boolean`) `autoPlay` signals whether you want to automatically start the remote playback of the media or not.

This component can handle Queues and can also provide controls for navigating through queue items

Remark 1*. To have all the metadata for a movie available when needed, you need to build and populate an instance of* `MediaInfo (defined in the Cast SDK)` *to represent a video. The fields that will be used in various places are:*

- metadata: use an instance of `MovieMetadata` that is populated with:
    - title
    - sub-title
    - studio
    - images
        - index 0: small size image used for notification, mini-controller and Lock Screen on JB
        - index 1: large image, used on the Cast Player page and Lock Screen on KitKat
- contentId (populate this with the URL pointing to the media)
- contentType: e.g. “video/mp4”
- streamType: typically of type `MediaInfo.STREAM_TYPE_BUFFERED`
- MediaTracks: Closed caption and tracks information, if available. See the section on Handling Tracks and Closed Captions below.

Remark 2*. It is very tempting to try to pass around* `MediaInfo` *objects from one activity to another. For example, if user is browsing a catalog and selects a video to be taken to the next activity that has detailed info on the media, it is ideal to use the* `MediaInfo` *that represents the selected media and pass that to the next activity. Unfortunately,* `MediaInfo` *is neither* `Serializable` *nor* `Parcelable` *so you cannot pass that in an intent; however, CCL uses a simple* `Bundle` *to wrap the* `MediaInfo`*, and provides two static utility methods* `Utils.bundleToMediaInfo()` *and* `Utils.mediaInfoToBundle()` *for conversion between the two types:*

```
    public static Bundle mediaInfoToBundle(MediaInfo info);
    public static MediaInfo bundleToMediaInfo(Bundle wrapper);
```

Volume Control

CCL provides an easy way for developers to handle remote volume control in their applications. Inside an activity, one can override `dispatchKeyEvent`:

```
@Override
public boolean dispatchKeyEvent(KeyEvent event) {
    return mCastManager.onDispatchVolumeKeyEvent(event, VOLUME_INCREMENT)
          || super.dispatchKeyEvent(event);
}
```

CCL will then handle the rest, i.e. it enables controlling the cast volume through the hard volume button on the phone inside that activity and also shows a visual volume bar when casting on supported versions. If `FEATURE_LOCKSCREEN` was also enabled during the initialization of the VideoCastManager, then CCL handles change of volume through the hard volume even if your app is not in front, is locked or even if the screen is off. In the above snippet, `VOLUME_INCREMENT` is a value that you set in your application and you need to also need to tell CCL about it by calling the following at an early stage, preferably at the initialization time:

```
mCastManager.setVolumeStep(VOLUME_INCREMENT);
```

The recommended value for `VOLUME_INCREMENT` when targeting a TV is 0.05 and for Audio-only devices is about 0.03.

Remark 3. *If you are targeting Android versions ICS and above, you do not need to overwrite the dispatchKeyEvent at all; MediaSessionCompat will handle this automatically but if you need to support GingerBread (GB), then this change is required and only works when app is in the foreground. In addition, the `VOLUME_INCREMENT` is only used on GB.*

Session Recovery (Reconnection)

To provide a seamless experience for users, we need to support the following scenario: assume that a client application is connected to a Cast device. If the user closes the application without calling an explicit disconnect, and later opens the same application, the client application needs to try to re-establish the connection to the same device as long as the Cast device is still running the same session that the user had started/joined earlier. If system has not yet killed your application, this does not require any effort but if your application has been killed by the system (or your connectivity was lost because you stepped outside of the wifi range, for example), some work needs to be done.

CCL can handle both of these cases with no effort on the application side. To enable the automatic session recovery when your app comes to the front, you need to enable this feature in `CastConfiguration` instance that is used in the initialization of the `VideoCastManager` instance. From that point on, CCL handles the rest. If, for some reason, you want to trigger the reconnection attempt, you can call the following API provided by CCL:

```
mCastManager.reconnectSessionIfPossible(timeoutInSeconds);
```

This method makes a best-effort attempt to reconnect to the same device if the same session is still running. The optional (int) `timeoutInSeconds` limits how long (in seconds) the whole reconnection effort should be attempted. If the parameter is not present, a 10 seconds timeout will be enforced. If during this timeout period the client was not able to establish a connection, the effort will be stopped.

There are more subtle, yet important, cases that are required to be handled. As an example, the user starts casting a long video and puts down her phone on the table; the phone goes to sleep and the wifi radio is turned off by the system. After a while, the user picks up her phone to pause the movie using the lock screen controls but since her phone had lost connectivity, she cannot take any immediate actions. Reconnection logic should handle such scenarios and as soon as the wifi connectivity is up again, it should try to connect to the same session and make the lock screen controls operational.

To handle those and other related cases that are explained in the Cast UI Checklist, CCL manages a long-running service `ReconnectionService` that can monitor the wifi connectivity and using the persisted connection and media information, tries to reconnect if possible. To enable the session recovery after recovering wifi connectivity, enable this feature in `CastConfiguration`. An application developer, however, can decide that in a specific situation, she doesn't want the reconnection to happen. The proper way to handle this is to clear the persisted data by calling `VideoCastManager.clearPersistedInfo(what)` (or `DataCastManager.clearPersistedInfo(what)`). The argument "`what`" can be one or a combination of different values that can restrict what portion(s) of the persisted information should be cleared, see the JavaDoc for more details.

Handling Tracks and Closed Captions

Handling tracks can be divided into three areas: API Support, Changing the Style, and UI Handling which will be discussed below.

API Support.

Cast SDK provides a number of APIs that can be used to define tracks, to associate them with a media (video) object, to set or update active tracks for a media and to set or update the style for

a text track[7]. The first step is to define a track: the class `MediaTrack` represents a track and using a builder pattern, on can define one or more tracks:

```
  MediaTrack englishSubtitle = new MediaTrack.Builder(1 /* ID */,
MediaTrack.TYPE_TEXT)
                .setName("English Subtitle")
                .setSubtype(MediaTrack.SUBTYPE_SUBTITLE)
                .setContentId("https://some-url/caption_en.vtt")
                // language is required for subtitle type but optional otherwise
                .setLanguage("en-US")
                .build();
```

Note that the first argument to the Builder is a unique ID that the application (not the SDK) defines and assigns to a track so that subsequent APIs can refer to this track.

The next step is to add one or more tracks to a video. This should happen before loading the media; any changes made to the tracks after loading the media will not take effect until the media is reloaded. To do that, include the tracks when you build your `MediaInfo` object:

```
      MediaInfo mediaInfo = MediaInfo.Builder(url)
                .setStreamType(MediaInfo.STREAM_TYPE_BUFFERED)
                .setContentType(getContentType())
                .setMetadata(getMetadata())
                .setMediaTracks(tracks) /* a list of tracks */
                .build();
```

Then you can load this media and set or update its stye. CCL provides the following wrapper methods:

```
/* Load a media and set its active tracks */
public void loadMedia(MediaInfo media, final long[] activeTracks, boolean autoPlay,
            int position, JSONObject customData)

/* Sets or updates the active tracks for the currently playing media */
public void setActiveTrackIds(long[] trackIds)

/* Sets or updates the style for the current text track */
public void setTextTrackStyle(TextTrackStyle style)

/* Gets the list of active track ids for the currently playing media */
public long[] getActiveTrackIds()
```

[7] See the section "Using the Tracks API" at https://developers.google.com/cast/docs/android_sender for more details

Changing the Style for Text Tracks

Users should be able to turn the captions on or off and to change the style of text tracks such as changing the size of the font, or the color of the background for captions. In Android KitKat and above, captions can be customized through Captions Settings, found under Settings -> Accessibility. Earlier versions of Android, however, do not have this capability. CCL handles this by providing custom settings for earlier versions and delegating to the system settings on KitKat and above. To take advantage of this feature, you first need to enable the support for this feature when building the `CastConfiguration`.

Then you can add an entry point into the Captions settings in your own application preferences; simply add the following snippet to the XML file that defines the preferences in your application (remember to replace `YOUR_APPLICATION_PACKAGE` with your own package name):

```
<PreferenceCategory
    android:title="@string/captions_settings">
    <PreferenceScreen
        android:key="CAPTION_KEY"
        android:title="@string/captions">
        <intent android:action="android.intent.action.VIEW"
android:targetClass="com.google.android.libraries.cast.companionlibrary.cast.tracks.C
aptionsPreferenceActivity"
            android:targetPackage="YOUR_APPLICATION_PACKAGE"/>
    </PreferenceScreen>
</PreferenceCategory>
```

You also need to add the following snippet to the `onResume()` of the `PreferenceActivity` of your application

```
mCastManager.updateCaptionSummary("CAPTION_KEY", preferenceScreen);
```

Note that the string `CAPTION_KEY` in this snippet should match the key used in the XML file above.

Adding these snippets provides an entry in the application's preferences that allows users to enable or disable captions and customize the style for all versions of Android. When the user makes a change in the captions settings, CCL will be notified automatically and will update the style of caption on the Cast device. In addition, the `VideoCastConsumer` has three callbacks that applications can listen to if they are interested in being notified of these changes:

```
public void onTextTrackStyleChanged(TextTrackStyle style);
public void onTextTrackEnabledChanged(boolean isEnabled);
public void onTextTrackLocaleChanged(Locale locale);
```

These parallel the callbacks that are available in Android KitKat+ through the system framework but CCL also extends them to the earlier versions.

Note that you need to define a certain activity in your manifest if you want to enable this feature; see the section on Manifest.

Setting Active Tracks from UI

Users need to have means to see the tracks that are available for media and be able to select one or more tracks. This is mainly the responsibility of the application but CCL makes it easier by providing a dialog that shows the available tracks and allows the user to select one or more tracks for a media. To open this dialog, use the following snippet:

```
TracksChooserDialog.newInstance(mMediaInfo)
        .show(mFragmentManager /* e.g. getSupportFragmentManager() */,"some_tag");
```

This dialog allows user to select a Text track or an Audio track. Upon making selections, CCL notifies any component that has registered to receive such information. For a component to do that, it has to implement the interface `OnTracksSelectedListener` and has to register itself with the `VideoCastManager` using the following api:

```
mCastManager.addTracksSelectedListener(OnTracksSelectedListener);
```

and remove itself when no longer interested:

```
mCastManager.removeTracksSelectedListener(OnTracksSelectedListener);
```

The `TracksChooserDialog` dialog extends `FragmentDialog` (from v4 support library), hence survives screen rotation and other configuration changes.

The Cast Player Screen also provides an icon that opens this dialog and allows user to select one or more tracks for the media that is playing on the cast device. Note that tracks support should be enabled in order for this icon to appear; if the media does not have any tracks, the icon will be grayed out (but visible) as long as the tracks support has been enabled.

Queues

Cast SDK provides support for a queue of media items (video or audio). Client applications should build a queue and then use any of the queue-related APIs that CCL provides to manage the queue, such as load, update, insert, delete, shuffle, … . The receiver SDK creates a queue

even if one single item is loaded using the normal load command (vs `queueLoad()`). CCL provides a number of callbacks to update the clients on changes to the queue such as `onMediaQueueUpdated()`. In addition, when you have a queue of video items with adaptive bitrate (such as a queue of HLS streams), then you can take advantage of "preload" feature which basically allows you to preload the next while the download of the currently playing item is completed; this allows a much smoother transition from one queue item to the next. CCL also provides a number of callbacks to inform the application of the preloading event, such as `onRemoteMediaPreloadStatusUpdated()`. See `VideoCastConsumer` for a complete list of available callbacks.

Single Custom Data Channel

For media-centric applications, CCL can also provides a single data channel (in addition to the standard media channel) to enable out-of-bound communication between the sender and receiver applications. For example, a sender application may need to provide a mechanism for its users to give a thumbs-up or thumbs-down when they are watching a content. These types of communications require a custom data channel.

To facilitate this, add a custom namespace while building `CastConfiguration`. You can choose a namespace of your choice and pass that as the last argument to that call. Then CCL will set up the sender part of the data channel and provides a single API for you to send a message to the receiver (`sendDataMessage()`), and two callbacks for you to remain informed when a message is received from receiver (`onDataMessageReceived()`) and when the send command encounters an error (`onDataMessageSendFailed()`). CCL also provides an API for you to remove your custom data channel (`removeDataChannel()`), but does the appropriate clean up for you when you disconnect from a device. Refer to the JavaDoc for the library for the documentation on these APIs and callbacks. Note that you would need to have a custom receiver to be able to send and receive messages using your own custom namespace.

Support for data-centric applications

If you are working with an application that is mainly data-centric and needs one or more custom data channels but no media data channel, you can initialize and use a different class in the library, the "`DataCastManager`". This class is the equivalent of the `VideoCastManager` and behaves similarly when you are setting it up. The initialization of this singleton is done by calling the following method:

```
    initialize(Context context, CastConfiguration options)
```

When building the instance of CastConfiguration, you can add as many custom namespaces as needed. Note that you can also add/remove namespaces later on by calling:

```
   addNamespace(String namespace); // to add a new namespace
```

```
   removeNamespace(String namespace); // to remove
```

The library takes care of setting up the data channel(s) for you. You can then use the following API to send a message on any of these channel(s):

```
void sendDataMessage(String message, String namespace)
```

This will send a message. Messages that are sent to the sender from the receiver can be captured by extending `DataCastConsumerImpl` class and overriding the following callbacks:

```
    public void onMessageReceived(CastDevice castDevice, String namespace, String
message) {}

    public void onMessageSendFailed(CastDevice castDevice, String namespace, long
messageId, int errorCode) {}
```

Hooking into Lifecycle Callbacks

The CCL library tries to handle most common scenarios but it also provides clients with access to events that can be useful in more advanced cases. For media-centric applications, clients can implement the `VideoCastConsumer` interface and register the implementation with the `VideoCastManager` instance. In case of data-centric apps, implement `DataCastConsumer` and register the implementation with the instance of `DataCastManager`.

If you refer to the documentation on these interfaces, you will find a long list of callbacks that range from connectivity success to application launch failure, etc. To make your life easier, the library introduces two no-op implementations of these interfaces: `VideoCastConsumerImpl` and `DataCastConsumerImpl`. Consequently, you can extend these classes and ignore all the interface methods except the ones that you are interested in and only override those methods. For example, here is how you can extend and register one:

```
        mCastManager.addVideoCastConsumer(new VideoCastConsumerImpl() {
            @Override
            public boolean onApplicationConnectionFailed(int errorCode) {
                // do what your application wants
                return true;
            }
        }
```

Remark 1. *In order to use the custom data channels properly, you need to implement the custom data related methods of these interfaces to at least receive messages from the receiver.*

Remark 2. *Most methods in these two interfaces have no return values. However, there are few of them that should return a* `boolean`*. These methods are related to failures in various stages (for example "launching application failed"). CCL can provide a standard error dialog for your application if your implementation of these methods returns* `true` *(that is the behavior of the default implementation classes, as well). If you want to provide your own error dialogs, make sure you override these methods and return* `false`*.*

Remark 3. *All the lifecycle callbacks are called on the Main (i.e. UI) Thread unless stated otherwise.*

Advanced Topics

Custom Notification

CCL provides a certain level of configurability for its built-in notification service. Clients, however, can register their own service to handle notifications. Steps to do that are simple: clients define a Service and the register that service with the `VideoCastManager`. This registration enables the library to manage the lifecycle and visibility of this service. Although a client application can define its notification service from scratch, it is recommended to extend the built-in `VideoCastNotificationService` to provide additional needed functionalities. For the latter, in most cases, clients would only need to override the "`build()`" method in the built-in service to construct their own notification. The protection level of a number of methods and fields in that class have been set to "protected" to enable an easy subclassing. Regardless of whether the client extends the built-in service or implements its own from scratch, that service needs to be registered with the `VideoCastManager`; this can be done when the `CastConfiguration` is built by calling the following method on `CastConfiguration.Builder`:

```
setCustomNotificationService(Class<? extends Service> customNotificationService)
```

Remember that you need to declare your service in your manifest and you also need to enable notification in `CastConfiguration` even if you use a custom notification service.

Live Streams

For live streams, the "pause" action should be interpreted somewhat differently. In general, if user starts playback after a pause in a live stream, player will not resume from the last point but instead will start from the current time in the live stream (in other words, there is no DVR functionality). As such, some applications prefer to use a "stop" icon instead of a "pause" when dealing with the live streams. Another difference is that a live stream cannot be "seeked" and most likely doesn't have a duration. CCL provides mechanisms for application developers to

handle these. First, one has to set the correct stream type on a media item so that CCL can recognize it as a live stream:

```
MediaInfo.Builder(url).setStreamType(MediaInfo.STREAM_TYPE_LIVE)....
```

For media items with the above stream type, CCL automatically uses a “stop” icon instead of a “pause” icon in all relevant places: in Mini Controllers, Notifications, Lock Screen, Media Router Controller Dialog and `VideoCastControllerActivity`. In addition, the `VideoCastControllerActivity` hides the seekbar and the duration for live streams. There are two technical details that you need to be aware of:

- For live streams, when user presses the “stop” button, the CCL library still sends the same “pause” command to the receiver. The reason for this is two fold: the receiver needs to provide its custom logic for handling this functionality, so it can interpret the “pause” command based on the type of media accordingly, and the second reason is that the standard “stop” command has a special functionality associated with it, for example it unloads the media and that is not the desired behavior for the live streams.
- When a live stream is “stopped”, receiver **must** send a media status update message back to the sender and should report the state as `MediaStatus.PLAYER_STATE_IDLE` with the reason `MediaStatus.IDLE_REASON_CANCELED`. For live streams, the CCL library will interpret this status appropriately.

Reconnection

One of the parameters that the Reconnection logic uses to control its behavior is the duration of the media that is playing on the cast device. When we are dealing with a live stream, there is no clear duration so the library uses a default value of “2 hrs”. This, however, can be changed if needed; call `VideoCastManager.setLiveStreamDuration(duration)` where `duration` should be specified in milliseconds.

Obtaining Authorization prior to playback

There are situations where an application needs to authorize a user before it allows the playback of a content. The CCL library has certain hooks and mechanisms in place to help with this process. Here is a list of steps:

1. Client applications need to provide an implementation of `MediaAuthService`[8]. The implementation is responsible for setting up the process internally but should not start the authorization process till its `start()` method is called.
2. When ready, the client application needs to call the following library method and pass the implementation of the interface to the framework:

   ```
   VideoCastManager.startCastControllerActivity(Context context, MediaAuthService
   authService)
   ```

[8] Despite the name, this has nothing to do with an Android Service; it can be a simple POJO.

3. Framework will build and pass an instance of `MediaAuthListener` to the implementation; this listener interface provides two callback methods, `onAuthResult()` and `onAuthFailure()`, using which, the implementation can communicate back with the library when its work is done, or if it encounters an error.
4. Framework will start the `VideoCastControllerActivity` and at an appropriate time calls the `start`() method of the `MediaAuthService` to start the authorization process. It also calls the `getMediaInfo()` on that interface to get the current media information. It is possible that at that point in the process, the `getMediaInfo()` returns a very limited information since, for example, the URL to the media may not yet be available until the authorization succeeds. The framework, however, expects to find the artwork assets in that early `MediaInfo` object so it can provide a background image for the `VideoCastControllerActivity` while the authorization process is happening. Note that the implementation has to provide a timeout (obtained through `MediaAuthService.getTimeout()`) so the framework can interrupt the process after a reasonable period.
5. When the authorization process is finished, the implementation has to call the `onAuthResult()` of the `MediaAuthListener` to inform the framework that the authorization process is completed; the arguments passed to this method will determine if authorization was granted or rejected. If the authorization process encounters an unrecoverable error, it has to call the `onAuthFailure()` of the `MediaAuthListener` to inform the framework.

Supporting Configuration Changes in `VideoCastControllerActivity`

A client application may need to support different configurations for the `VideoCastControllerActivity` and provide different layouts for each configuration, for example a landscape and a portrait layout. To provide continuity during the configuration changes (especially if a pre-authorization or a normal load is happening), the framework uses a `Fragment` (`VideoCastControllerFragment`) to maintain the state and to handle the work; this fragment does not contribute any UI; the UI is completely handled by the `VideoCastControllerActivity`. A common approach to customize the UI is to copy the layout resources from the library to your project (layout is defined in `cast_activity.xml`) and modify the layout in your own project (without changing the IDs of the components). In some cases, this is still not as flexible as an application needs. To further accommodate that, the CCL library allows you to define your own activity and yet use the `VideoCastControllerFragment` to manage the lifecycle and interaction with a cast device. In order for that to work, your activity should implement `VideoCastController` interface (`VideoCastControllerActivity` itself is an implementation of that interface). Please take a look at the `VideoCastControllerActivity` to see how the `VideoCastControllerFragment` needs to be called.

Configuration and Setup

Manifest file

Your `AndroidManifest.xml` file needs to include certain elements and metadata for the library and Cast SDK proper operation. Here is a list of requirements:

- Minimum SDK version: the minimum SDK version that the library works with is 10 (GingerBread 2.3.3).
- Permissions: using the cast functionality does not require any additional permissions. If, however, you have enabled Wifi Reconnection, then you need to add the following two permissions:

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

- Application Metadata: like any application that uses the Play Services, add the following metadata to your application declaration:

```
<meta-data android:name="com.google.android.gms.version"
         android:value="@integer/google_play_services_version" />
```

- Makes sure your application's "theme" is correctly set based on the min SDK version. For example, you may need to use a variant of `Theme.AppCompat` or one that inherits from these themes. Note that this is not a requirement for Cast, per se.
- Declare the VideoCastControllerActivity:

```
    <activity
android:name="com.google.android.libraries.cast.companionlibrary.cast.player.V
ideoCastControllerActivity"
            android:screenOrientation="portrait"
            android:label="@string/app_name"
            android:launchMode="singleTask"
            android:parentActivityName="*MY_PARENT_ACTIVITY*"
            android:theme="*AN_OVERLAY_THEME*">
            <meta-data
                android:name="android.support.PARENT_ACTIVITY"
                android:value="*MY_PARENT_ACTIVITY*" />

            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
            </intent-filter>
        </activity>
```

There are a few things that you need to set in the above snippet. Assume that user clicks on your application's notification from the status bar while casting. The desired behavior is to open your application and direct user to the `VideoCastControllerActivity`. However, this is not the main launcher activity of your application, so if the user were to press the back button, she should not be sent out of the application. The correct behavior would be to send the user to the parent activity; the one that in a normal execution of your application would show up when user goes back from the `VideoCastControllerActivity`. This special behavior can be achieved by creating a "back stack" for the `PendingIntent` in the notification service. CCL can handle this for you if you declare in the manifest what activity is the correct parent activity; you need to enter the name of that activity (possibly a fully qualified name) in the two places in the above snippet where the `*PARENT_ACTIVITY*` is used[9]. In addition, the `VideoCastControllerActivity` is best presented if you use an overlay Action Bar theme; the CastVideos reference app defines one (`DemocastOverlay`) that you can copy and use, or you can define your own theme; it should replace the `*AN_OVERLAY_THEME*` in the above snippet.

- Declare the following receiver:

```
<receiver
android:name="com.google.android.libraries.cast.companionlibrary.remotecontrol.VideoI
ntentReceiver" >
    <intent-filter>
        <action android:name="android.media.AUDIO_BECOMING_NOISY" />
        <action android:name="android.intent.action.MEDIA_BUTTON" />
        <action
android:name="com.google.android.libraries.cast.companionlibrary.action.toggleplaybac
k" />
        <action
android:name="com.google.android.libraries.cast.companionlibrary.action.stop" />
    </intent-filter>
</receiver>
```

This receiver is called when user interacts with the `VideoCastNotificationService` where it broadcasts a message that this receiver catches and handles appropriately. Note that the actions that are defined in the above snippet should match the actions that you want to enable in your notification (and were configured when building the instance of CastConfiguration), here is a list of possible values used in the name attribute of the <Action/> elements inside the above receiver:

```
Fast Forward: "com.google.android.libraries.cast.companionlibrary.action.forward"
Rewind:       "com.google.android.libraries.cast.companionlibrary.action.rewind"
Play/Pause:   "com.google.android.libraries.cast.companionlibrary.action
```

[9] See http://developer.android.com/guide/topics/ui/notifiers/notifications.html#NotificationResponse for more information

```
                                                                    .toggleplayback"
Skip Next:    "com.google.android.libraries.cast.companionlibrary.action.playnext"
Skip Prev:    "com.google.android.libraries.cast.companionlibrary.action.playprev"
Disconnect:   "com.google.android.libraries.cast.companionlibrary.action.stop"
```

- Declare the following service (for `VideoCastNotificationService`):

```
<service
android:name="com.google.android.libraries.cast.companionlibrary.notification.VideoCa
stNotificationService"
    android:exported="false" >
    <intent-filter>
        <action
android:name="com.google.android.libraries.cast.companionlibrary.action.notificationv
isibility" />
    </intent-filter>
</service>
```

- Declare the following service (for `ReconnectionService`) if Wifi Reconnection has been enabled:

```
<service
android:name="com.google.android.libraries.cast.companionlibrary.cast.reconnection.Re
connectionService"/>
```

- If you want to enable Captions in the preferences, declare the following activity in your manifest:

```
<activity

android:name="com.google.android.libraries.cast.companionlibrary.cast.tracks.C
aptionsPreferenceActivity"
    android:label="@string/action_settings" >
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
    </intent-filter>
</activity>
```

Configurable Messages

Here is a list of strings defined in the strings.xml file that the application client can override in its own strings.xml, please consult that file for an up-to-date list:

```
<string name="ccl_version">2.0</string>
<string name="ccl_media_route_menu_title">Play on&amp;#8230;</string>
<string name="ccl_live">Live</string>
<string name="ccl_cancel">Cancel</string>
<string name="ccl_on">On</string>
<string name="ccl_off">Off</string>
<string name="ccl_info_na">Information Not Available</string>
<string name="ccl_pause">Pause</string>
<string name="ccl_disconnect">Disconnect</string>
<string name="ccl_none">None</string>

<!-- Used in Error Dialog -->
<string name="ccl_ok">OK</string>

<!-- Title of the Error Dialog -->
<string name="ccl_error">Error</string>

<!-- Shown in VideoCastController Activity -->
<string name="ccl_casting_to_device">Casting to %1$s</string>
<string name="ccl_loading">Loading&amp;#8230;</string>

<!-- Used in Router Controller Dialog -->
<string name="ccl_no_media_info">No media information available</string>

<!-- onApplicationConnectionFailed() errors -->
<string name="ccl_failed_to_launch_app">Failed to launch application</string>
<string name="ccl_failed_app_launch_timeout">The request to launch the application
has timed out!</string>
<string name="ccl_failed_to_find_app">The application you are trying to launch is not
available on your Cast device</string>

<!-- Error messages for the play.pause in mini player -->
<string name="ccl_failed_to_play">Failed to start the playback of media</string>
<string name="ccl_failed_to_stop">Failed to stop the playback of media</string>
<string name="ccl_failed_to_pause">Failed to pause the playback of media</string>

<!-- Failures -->
<string name="ccl_failed_to_connect">Could not connect to the device</string>
<string name="ccl_failed_setting_volume">Failed to set the volume</string>
<string name="ccl_failed_no_connection">No connection to the cast device is
present</string>
<string name="ccl_failed_no_connection_short">No connection</string>
<string name="ccl_failed_no_connection_trans">Connection to the cast device has been
lost. Application is trying to re-establish the connection, if possible. Please wait
for a few seconds and try again.</string>
<string name="ccl_failed_perform_action">Failed to perform the action</string>
<string name="ccl_failed_status_request">Failed to sync up with the cast
device</string>
<string name="ccl_failed_seek">Failed to seek to the new position on the cast
device</string>
<string name="ccl_failed_receiver_player_error">Receiver player has encountered a
severe error</string>
<string name="ccl_failed_authorization_timeout">Authorization timed out</string>
```

```
<string name="ccl_failed_to_set_track_style">Failed to update the captions
style.</string>
```

Setting up CCL for your application

To add this library to your application as a dependency, follow these steps.

a. Clone the CastCompanionLibrary-android under the name CastCompanionLibrary, parallel to your project (or create a symbolic link to it parallel to your project), i.e. your project and CastCompanionLibrary should share the same parent:

```
$ git clone https://github.com/googlecast/CastCompanionLibrary-android.git
CastCompanionLibrary
```

b. In the root of your project, edit the file “settings.gradle” and add the following two lines:

```
include ':CastCompanionLibrary'
project(':CastCompanionLibrary').projectDir = new File('../CastCompanionLibrary/')
```

c. Edit build.gradle in your main module (commonly named “app” module) and add the highlighted dependency:

```
dependencies {
   ...
   compile project(':CastCompanionLibrary')
}
```

Then you will see a new module in your Android Studio, called CastCompanionLibrary, that is the source for the library and your application is ready to use it.