Deezer iOS SDK User Guide

Contents

This new version of the Deezer SDK is a total departure from the previous version. This means that previous integrations you may have made will not work with this beta version and any new release of the Deezer SDK (except for session management). That said, this new beta version brings a lot of improvements and is easier to integrate than its previous incarnation.

1 Main changes

Hide details of Deezer's web service API
The new SDK hides access to the Deezer API. The knowledge of this API is still somewhat necessary to know which properties you can query on the different objects. But you no longer have to know the endpoints to call. Instead, you now must use the model objects provided by the SDK to query information on Deezer objects. The SDK handles the eventual caching of data and network requests. Take a look at the section concerning model objects for more information.

Networking stack
Deezer provides you with a set of networking utilities. These are mainly used by the SDK for its own network management, but you can also use it in your application for your own benefit. Please read the corresponding network stack section.

Enhanced player
The initial version of the iOS SDK only provided the means to read one track at a time. The integration was required to manage eventual chaining of tracks on its own, creating one player instance for each track to play.

The player now can play collections of tracks:

- albums
- playlists
- radios

The integration just has to give the right model object to the player and let it manage the playing of all elements. Please refer to the section dedicated to the player for more information on this.

Support for playlists', artists' radio and *Flow*
In addition to themed radio channels, the player now support playlist and artist radio and Flow (personalised user radio). You can read more about this feature under the DZRRadioStreamable and DZRFlowable sections.

Support for private and uploaded tracks
Finally the player is able to play the tracks that you uploaded to Deezer as well as the tracks uploaded by the artists (*D4A*).

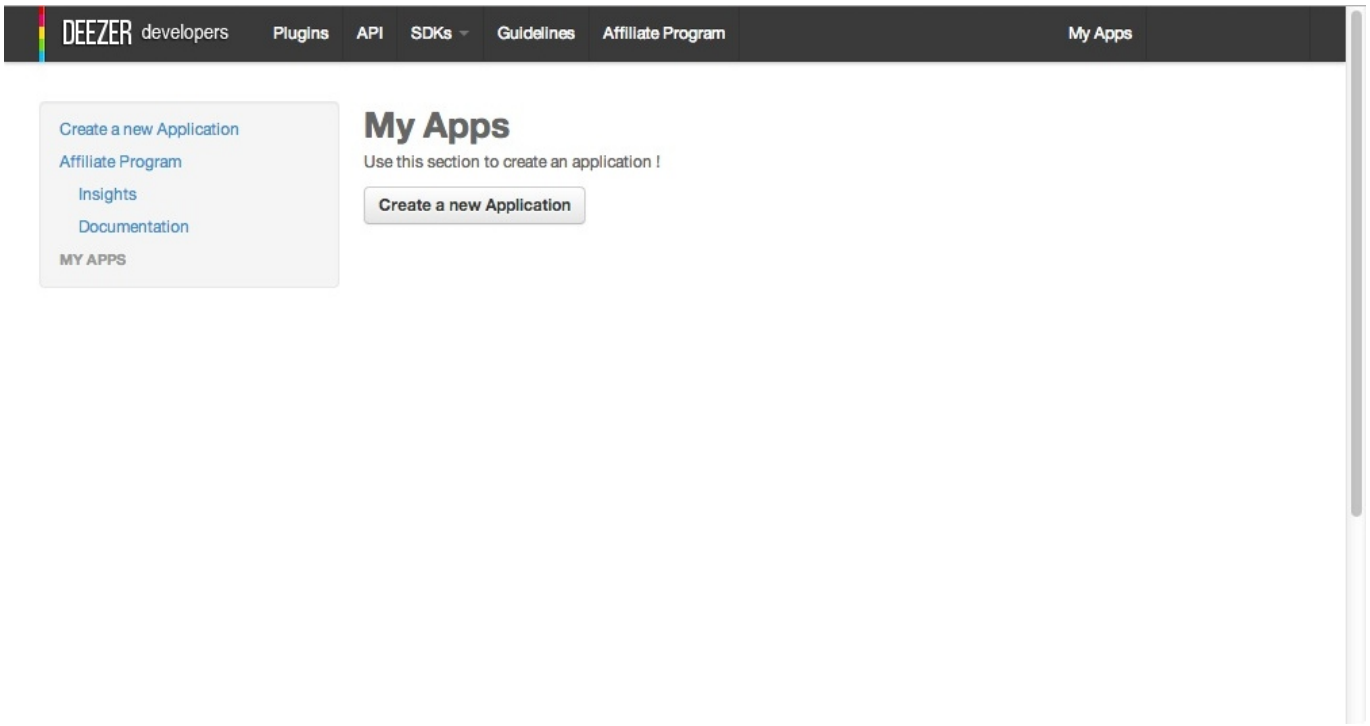
2 Quick Start

In this section, we provide you with a step by step guide to get you up and running with the SDK as quickly as possible.

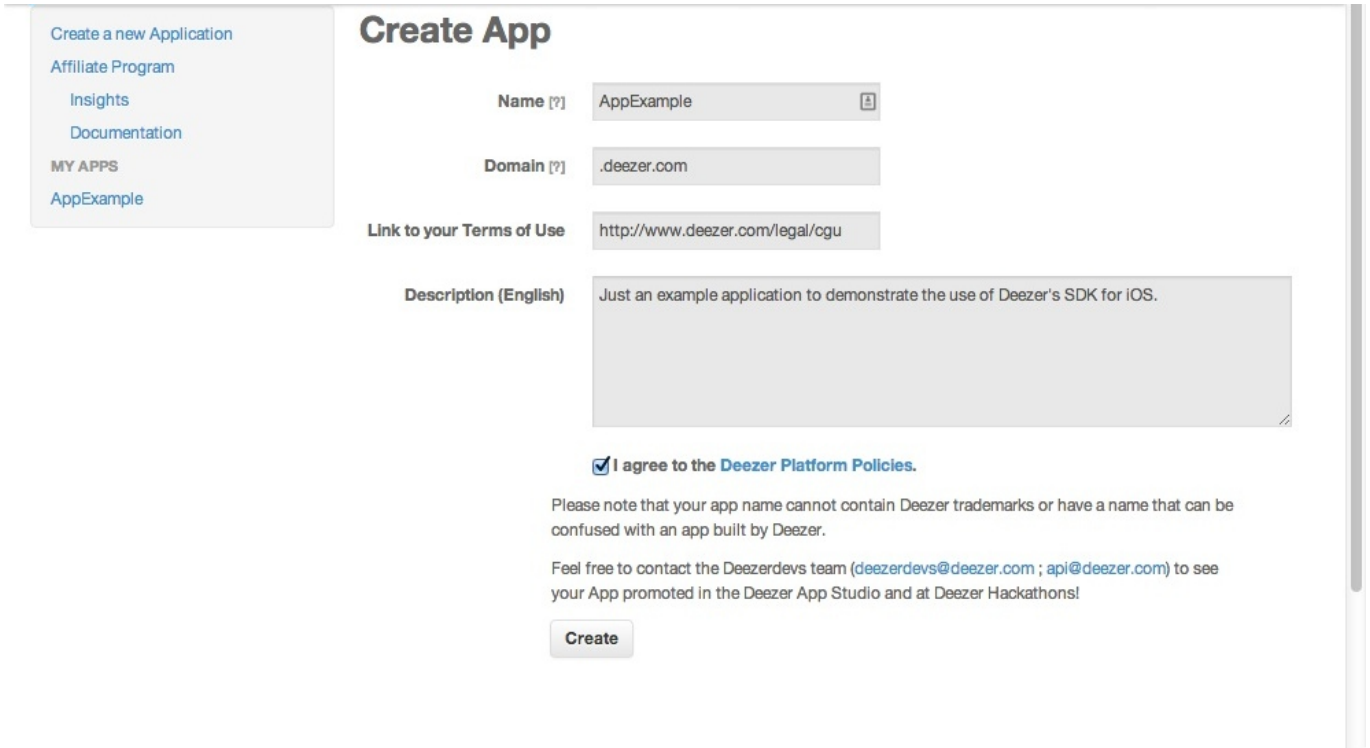
2.1 Registering you application with Deezer

The first step is to register your application with Deezer. This will let you obtain an *appid* which is necessary to initialise the Deezer iOS SDK and configure the Deezer application to allow users to sign in.

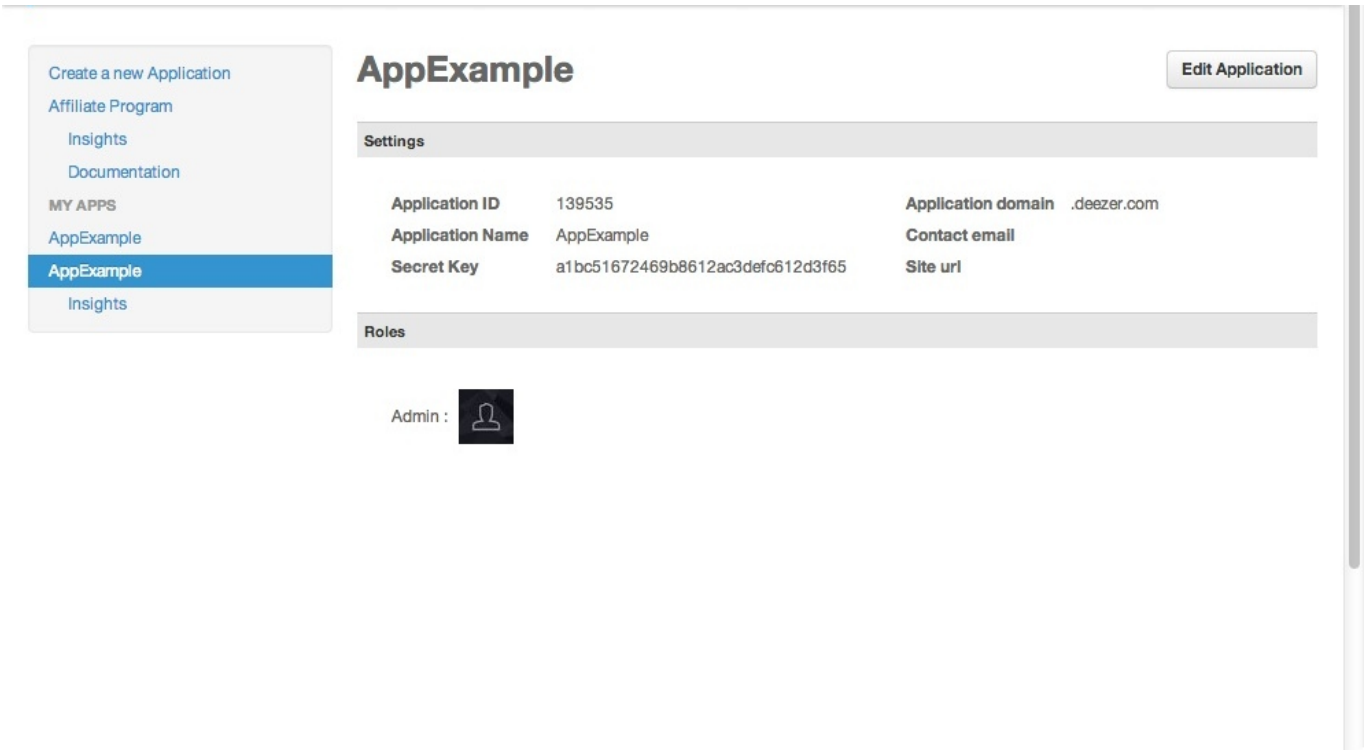
To register your application, you will need a Deezer account. Please go to http://developers.Deezer.com/, log in into your account or create one if needed. Next, click on the *My Apps* button next to your login, to the right of the tab bar.



A screen looking like the one just above will be presented to you. If you already created your application, click on its name in the left sidebar (it will appear under *MY APPS*) and configure it as described later in this section. Otherwise, click the button *Create a new Application* to create one.

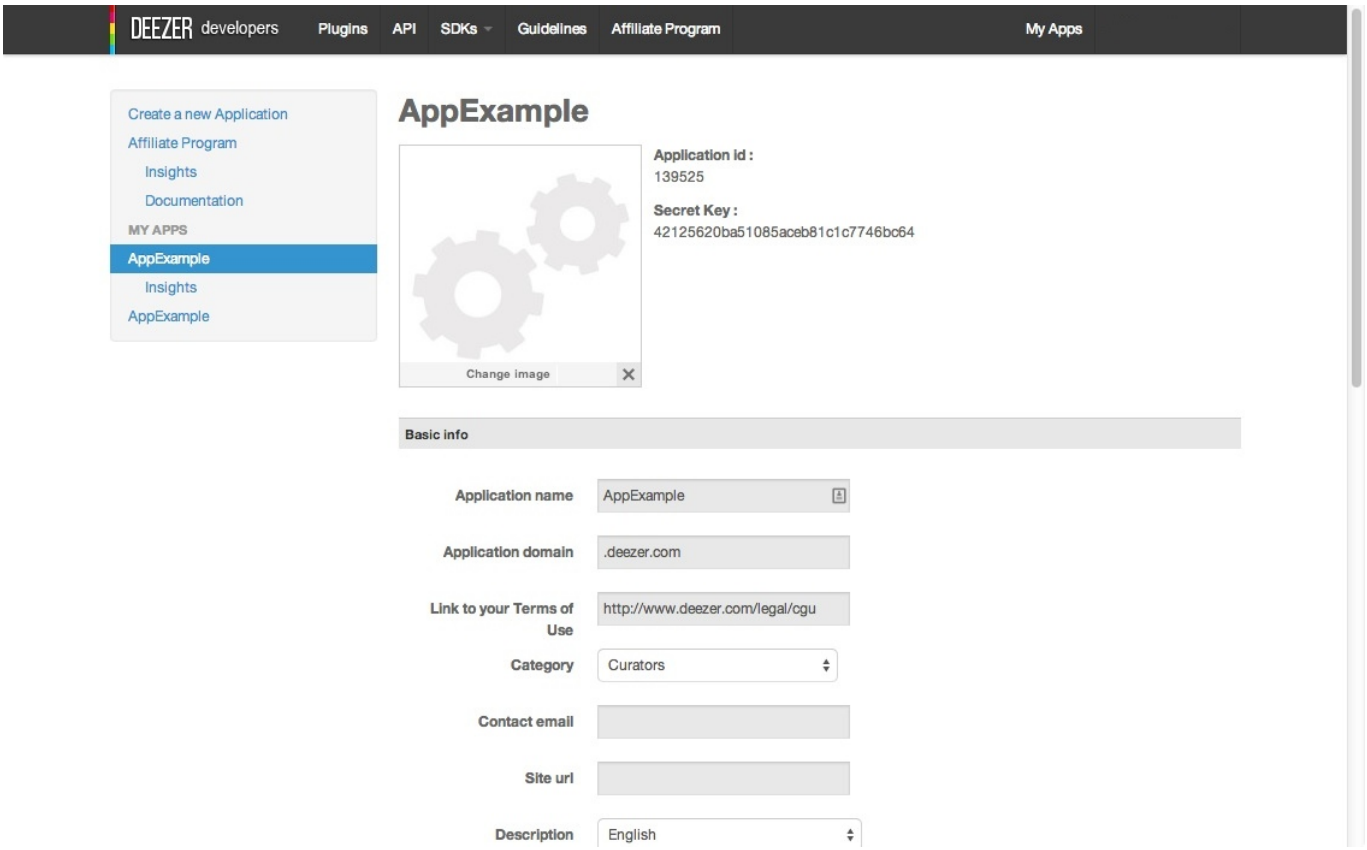


In order to create your application complete the required fields, tick the *"I agree to te Deezer Platform Policies"* after having read the said document and finally click *Create* to validate the form.



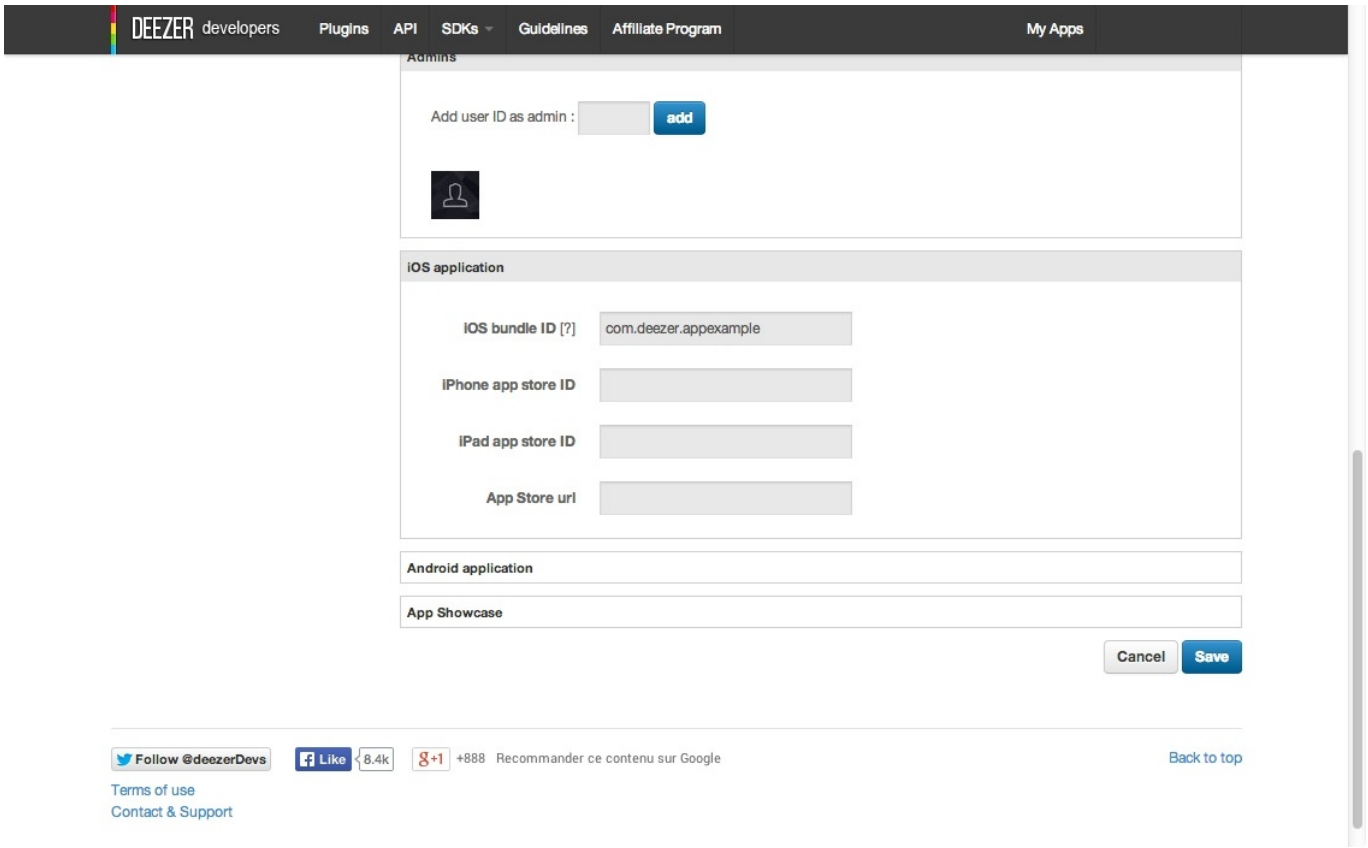
Your Deezer application has now been created. Note that it appears under the *MY APPS* section in the left sidebar.

The process is not finished yet though. You now need to configure the iOS section of your Deezer application to be able to let users log in with their Deezer account. In order to proceed, click the *"Edit Application"* button in the top right corner of the screen.

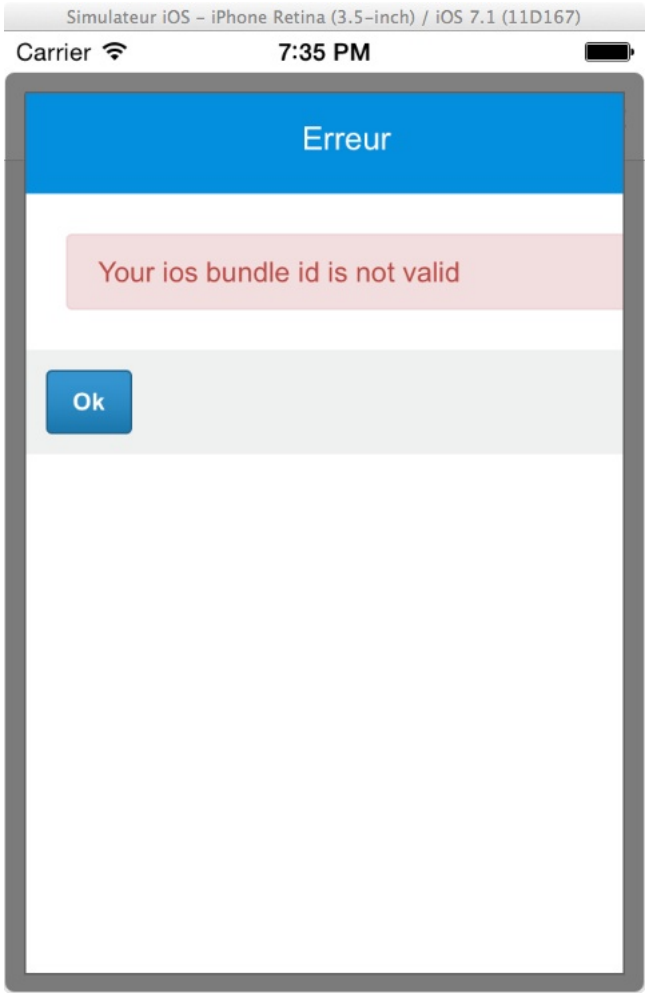


You are now presented with the interface to edit your Deezer application's details and configure it for the different supported

platforms. Please take the time to complete all necessary fields in the *"Basic info"* section. Next, scroll down to the bottom of the page and click on *"iOS Application"* to reveal iOS specific configuration.



Here you need to at least complete the *iOS Bundle ID* field. Without this information, the SDK will initialise properly but all authentication attempts will be rejected.



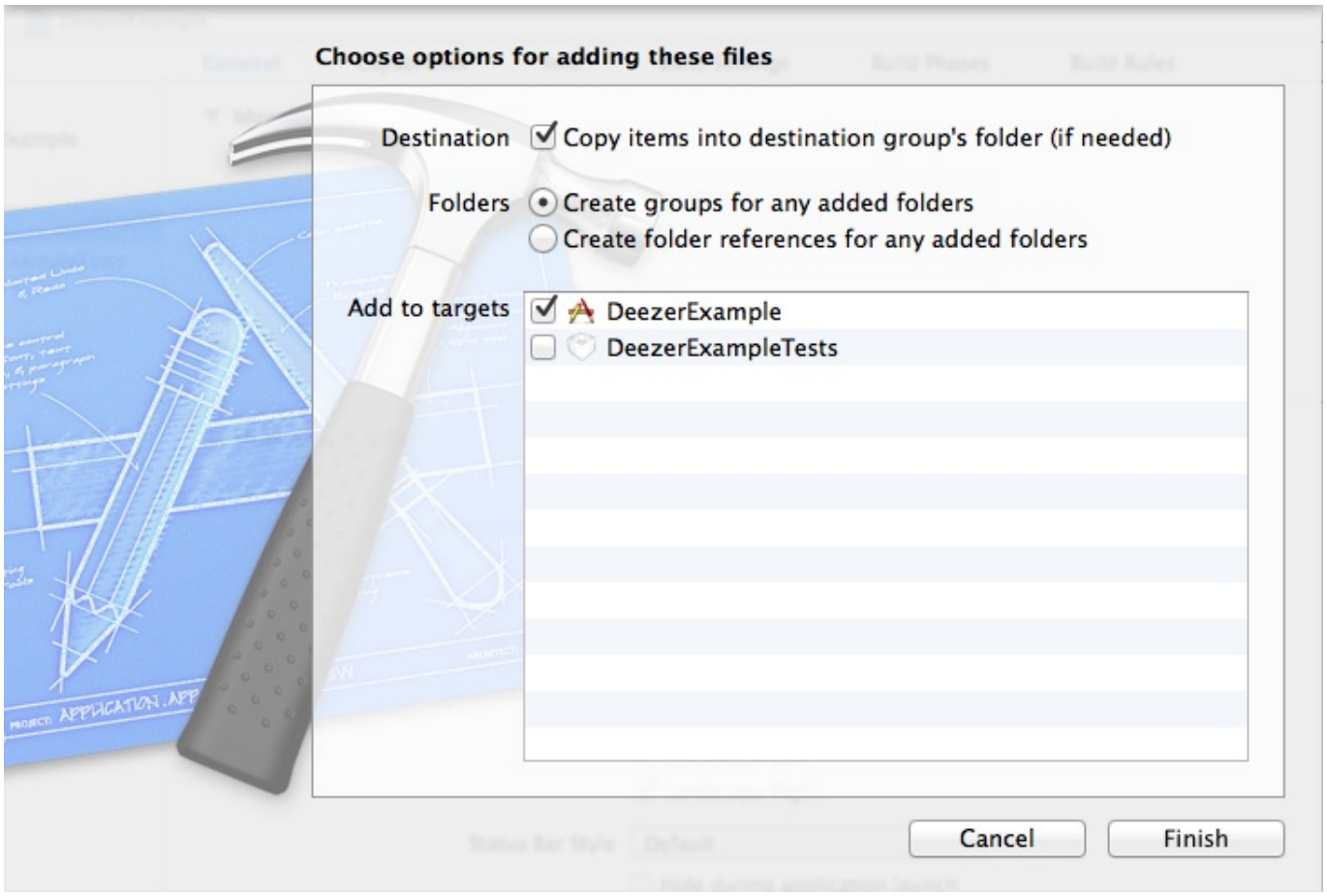
Your application is now properly registered with Deezer and you can proceed to configure your XCode project.

2.2 Configuring your XCode project

To configure you project, you will have to first download and unzip the SDK archive. Inside you should find:

- The static library `libDeezer.a`
- An `include` folder containing the header files
- A resource bundle `DZRResources.bundle`

Drag and drop the folder obtained from unzipping the SDK just under your application target's name in the XCode project navigator. XCode will display a dialogue. Make sure your application's target is ticked in the section *Add to target* and that the *Copy items into destination group's folder (if needed)* is activated.



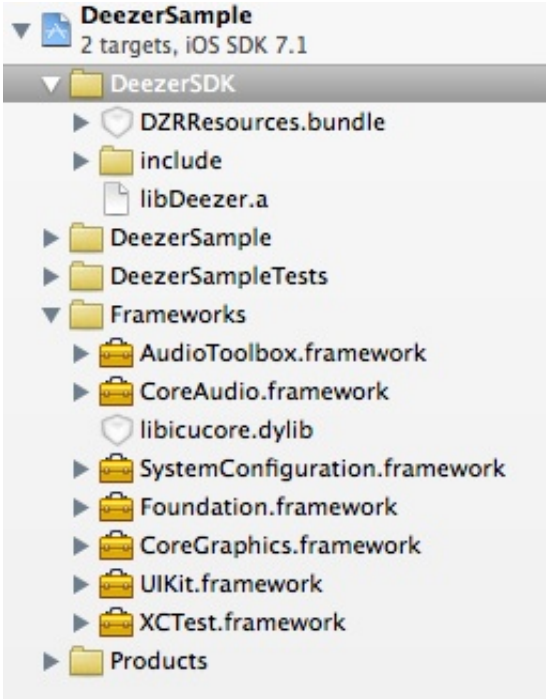
Make sure to include the resource bundle
Make sure that you include the resource bundle into your application's target. The SDK will otherwise complain and exit the application at run time.

Next, you need to link the application's target with the following:

- CoreAudio.framework
- AudioToolbox.framwork
- SystemConfiguration.framework
- libicucore.dylib
- MediaPlayer.framework

And add the option `-all_load` to the *Other Linker flags* target's configuration.

When you have finished configuring your project, the project view should look similar to the next capture.



Last but not least, check that the bundle ID of your application matches the one you set when configuring your application on Deezer's developer website.

2.3 Initializing the SDK

Initializing a `DeezerConnect` class **is the very first thing you should do in order to use the Deezer SDK for iOS**. To do this, send the `-[DeezerConnect initWithAppId:andDelegate:]` message to an allocated object `DeezerConnect`. The first parameter is the *appId* that you got from your Deezer application page in your Deezer developer account. For the second argument, you must provide a delegate object to the `DeezerConnect` implementing the `DeezerSessionDelegate` protocol. It is through this protocol that the `DeezerConnect` object will later inform your application of the changes in login status.

Finally, you must associate the `DeezerConnect` instance with the network stack. To do so, you need to set the `dzrConnect` property of your default `DZRRequestManager` (see the Network stack section for more information on `DZRRequestManager`). This **root** is generally the shared instance of the `DZRRequestManager`:

Associating the DeezerConnect object with the network stack

```
[[DZRRequestManager defaultManager] setDzrConnect:deezerConnect];
```

3 Session Management

No significant modification was made to the session management in this release. You can skip this section entirely if you use the session management from the previous SDK. The only change is to the management of network requests. You no longer have access to these methods since the Deezer model objects take care of this. See the section about model objects to learn more on this.

To let your users log in with their Deezer account, you should use a `DeezerConnect` instance. You should have already instantiated a `DeezerConnect` to initialize the SDK. At this time you provided a delegate to the `DeezerConnect` instance. This delegate will be notified of the login status change during the login process.

```
@protocol DeezerSessionDelegate <NSObject>
@optional
- (void)deezerDidLogin;
- (void)deezerDidNotLogin:(BOOL)cancelled;
- (void)deezerDidLogout;
@end
```

To start the login process, you must send the `-[DeezerConnect authorize:]` message passing in an array of strings representing the permissions your application is requesting. The SDK provides a set of constant strings matching the permissions outlined in the Deezer web service documentation:

- DeezerConnectPermissionBasicAccess
- DeezerConnectPermissionEmail
- DeezerConnectPermissionOfflineAccess
- DeezerConnectPermissionManageLibrary
- DeezerConnectPermissionDeleteLibrary
- DeezerConnectPermissionListeningHistory

The login process depends on the context:

- If a user is not logged into Deezer, they will be presented with a login screen that the user has to complete.
- Then Deezer's SDK will present a screen explaining to the user about the permission request from your application and asking for their approval.
- If a user is already logged in, but your app is requesting additional permissions, only the permissions screen will be presented.

The login process will continue and the `DeezerConnect` nstance will inform you by calling your implementation of `DeezerSessionDelegate`'s protocol. When the user completes the login process, your application will be notified by a call to `-[id<DeezerSessionDelegate> deezerDidLogin:]` method on the delegate. In your implementation of this method, you can query the `DeezerConnect` instance for login information such as:

- the access token provided by the OAuth API: `-[DeezerConnect accessToken]`.
- the the token's expiration date: `-[DeezerConnect expirationDate]`.
- the Deezer's user identifier: `-[DeezerConnect userId]`.

Saving credentials for later use
identifier, token and expiration) in the application's keychain for later use. The next time the user launches the application, set these values back on the `DeezerConnect` instance and test their validity by sending the `-[DeezerConnect isSessionValid]` message to this same object. If the session is still valid, do not ask the user to log in again.

You must also provide a means for the user to logout from their Deezer account. You can do so by sending the `-[DeezerConnect logout]` message the your `DeezerConnect` instance.

Erasing credentials at logout time
If you have saved the credentials for reuse at login time, you **should** erase them at this point, as is expected by your user. Keeping stale credentials can be both a usability and trust issue.

---

4 Network stack

The network stack provided by the Deezer SDK is tailored to work well with the Deezer web service, but perhaps it can also suit your needs. This stack is classical in its architecture, looking a lot like other popular networking frameworks around the iOS community. However it offers some features that may be of interest.

4.1 Network Request

All the network requests issued by the Deezer SDK pass through the `DZRRequestManager`. In order to schedule requests to the network, you first have to instantiate an object from the `DZRNetworkRequest` class hierarchy. `DZRNetworkRequest` is the base class and only requires a handful of information in order to be configured.

DZRNetworkRequest

```
@interface DZRNetworkRequest : NSObject<DZRCancelable>          // ❶
@property (nonatomic, strong) NSURL *URL;                       // ❷
@property (nonatomic, assign) DZRRequestPriority priority;      // ❸

@property (nonatomic, copy) void (^dataCompletionBlock)
    (NSData* response, NSError* error);                         // ❹

- (id)initWithURLString:(NSString*)urlString;                  // ❺
- (id)initWitURL:(NSURL*)URL;                                  // ❻
@end
```

❶ Network request are cancelable, `DZRCancelable` only defines a single message: `cancel`. You can cancel a request before it is sent to the network, or when it is loading. Cancelling after the request is completed is obviously a no-op.

❷ This is the URL you want to request.

❸ You can set a priority to a request. Requests with higher priority will be sent before the ones with lower priority. See the section on request priorities for further explanation of the values you can set for this property.

❹ This is a callback that will be called when the request is complete, there is an error, or it is cancelled. Depending on the context, the `response` or `error` will be set appropriately.

❺, ❻ Initialisers.

The only essential information you must provide is the `URL`. Everything else is optional and depends on the use case.

4.1.1 The result callback `dataCompletionBlock`

If you expect to have response data back from the server, you should provide a `dataCompletionBlock` callback. It will be called when the request finishes (with or without error). If the request completes without any errors and its response contains a body, you will be sent a `NSData` instance containing the bytes for the response, the `error` parameter being `nil`. On the other hand, if the request encountered an error or was cancelled your callback will be called with an `error` set and `nil` for the data.

4.1.2 Request priorities

You can set the `priority` property of the request. Three values are possible:

`DZRRequestPriorityLow`
: which is the lowest priority. The requests using this priority are not guaranteed to be issued quickly and can be delayed until the manager can batch them or a request with a higher priority is scheduled. You can use this priority to send some tacking information, for example.

`DZRRequestPriorityNormal`
This is the default priority. Requests configured with this priority will be sent as soon as possible but can be delayed when the manager is busy handling higher priorities or prior requests. You can generally let the request `priority` be set to this value most of the time.

`DZRRequestPriorityHigh`
This is the highest priority you can specify for a request, and will be processed by the manager as a priority. If no slot is currently available for a request with this priority, the manager can send a lower priority request back in the queue in order to free up a slot. You should use this priority only for times when you need information urgently from the server.

4.1.3 Specialized network requests

The network stack provides a handful specialised `DZRNetworkRequest` subclasses:

```
@interface DZRJSONRequest : DZRNetworkRequest     // ❶
@property (nonatomic, copy) void(^JSONCompletionBlock)
    (id JSON, NSError* error);
@end

@interface DZRJSONPostRequest : DZRJSONRequest    // ❷
@property (nonatomic, strong) NSDictionary *postContent;
- (id)initWithURLString:(NSString *)urlString
         andPostContent:(NSDictionary*)postContent;
- (id)initWithURLString:(NSString *)urlString
             andConnect:(DeezerConnect *)connect
            postContent:(NSDictionary*)postContent;
@end

@interface DZRJSONDeleteRequest : DZRJSONRequest  // ❸
- (id)initWithURLString:(NSString *)urlString
              andParams:(NSDictionary*)params;
- (id)initWithURLString:(NSString *)urlString
             andConnect:(DeezerConnect *)connect
                 params:(NSDictionary*)params;
@end

@interface DZRImageRequest : DZRNetworkRequest    // ❹
@property (nonatomic, copy) void(^imageCompletionBlock)
    (UIImage* image, NSError* error);
@end
```

❶ This request can be used to request JSON payloads from a server. You should provide a `JSONCompletionBlock` that will be called just after the `dataCompletionBlock` with the parsed contents.

❷ In addition to requesting a JSON payload, you can also send some payloads to the server by using this `DZRNetworkRequest` subclass. Just set your payload to the `postContent` The content will be in the form of an encoded URL and will be sent to the server. The server is expected to respond with a JSON payload.

❸ This subclass is used to issue a `DELETE` request to the server. You can provide additional information if needed.

❹ This subclass is used to download images from a server. Specifically an `imageCompletionBlock` to get the image. This completion block is called after the `dataCompletionBlock`.

4.2 Scheduling network requests

After constructing your request, you must submit it to a request manager. The interface of `DZRRequestManager` is concise:

Request Manager

```
@interface DZRRequestManager : NSObject
+ (instancetype)defaultManager;                                    // ❶

@property (nonatomic, strong) DeezerConnect *dzrConnect;           // ❷
- (void)addRequest:(DZRNetworkRequest *)request;                  // ❸
- (void)cancel;                                                   // ❹

- (DZRRequestManager*)subManager;                                 // ❺
- (DZRRequestGroup*)groupRequests:(void(^)(DZRRequestManager* requestManager)) ↩
    groupTransaction;
@end
```

❶ This is the singleton method returning the default request manager.

❷ A `DeezerConnect` bject from which the manager can retrieve credentials if it needs to interact with Deezer's web services. Sub-managers inherit their parent's `dzrConnect`. o assigning a `DeezerConnect` instance to the default manager means that all sub-managers of the application will be able to use it.

❸ Schedules a request on this manager. The request is queued and will be issued by the manager when a slot becomes free.

❹ You can cancel a manager. This is useful for sub-managers (see below) but not advisable for the default manager. Cancelled sub-managers are unusable after receiving this message (all current and future connections are automatically cancelled). The default manager is unique since only current requests are cancelled. If you later queue new requests, they will be managed normally by the default manager.

❺ These two messages are used to create a sub-manager. Sub-managers can be used instead of the default manager to allow a more coarse-grained control on requests. See the next section on manager trees for more information.

4.2.1 Manager trees

It is advisable to organise managers in a tree with the default manager at the root of this tree. This allows a more convenient way to cancel a group of requests (and thus helps the management of network resources).

An example could be to tie the lifetime of a manager (together with its managed request and sub-managers) to the lifetime of a view controller. When the holder of the manager (view controller) is destroyed, it cancels its manager which in turn cancels its scheduled network requests and all its sub-managers. In the case of a view controller, this mechanism alleviates the management of (many) short requests to display information on the controller's view. Another classical example of this technique is to use a sub-manager for requesting resources like images in table view cells

Coupling view controller and network manager

```
- (id)init
{
    if (self = [super init]) {
        self.manager = [[DZRRequestManager
                         defaultManager] subManager]; // ❶
    }
    return self;
}

- (void)dealloc {
    [self.manager cancel];                          // ❷
    self.manager = nil;
}

- (IBAction)actionWastapped:(id)sender
{
    [self.dzrObject
     valueForKey:@"id"
     withManager:self.manager                       // ❸
```

```
     callback:^(NSString *identifier, NSError *error) {}];
}
```

❶ At the time of initialisation of your view controller, create a sub-manager. Here we create a direct descendant of the default manager. You could also pass a parent manager to the initialisation and use it to create your sub-manager, chaining your sub-manager to the sub-manager of your parent view controller, for example.

❷ When the view controller is being freed, cancel the tied manager to be sure that the requests scheduled in the view controller implementation will be cancelled and not pollute the network stack unnecessarily.

❸ On occasions when you need to schedule a network request in the implementation of your view controller, use the tied sub-manager.

Using a sub-manager to request thumbnail images

```
#pragma mark UIScrollViewDelegate

- (void)scrollViewDidEndDecelerating:(UIScrollView *)scrollView
{
    self.illustrationManager = [self.manager subManager];                // ❶
    [[objectsView visibleCells]
      enumerateObjectsUsingBlock:^(UITableViewCell *cell, NSUInteger idx, BOOL *stop) {
        DZRObject *o = [self.data objectAtIndex:[objectsView indexPathForCell:cell].row];
        if (o.isIllustratable) {
            UIImage *cachedImage = [self.imageCache objectForKey:o];
            __weak NSCache *cache = self.imageCache;
            if (!cachedImage) {
                [(DZRObject <DZRIllustratable> *) o
                 illustrationWithRequestManager:self.illustrationManager // ❷
                 callback:^(UIImage *illustration, NSError *error)
                 {
                     if (illustration) {
                         [cache setObject:illustration forKey:o];
                         cell.imageView.image = illustration;
                         [cell setNeedsLayout];
                     }
                 }];
            }
            else {
                cell.imageView.image = cachedImage;
                [cell setNeedsLayout];
            }
        }
    }];
}

- (void)scrollViewWillBeginDragging:(UIScrollView *)scrollView
{
    [self.illustrationManager cancel];                                   // ❸
    self.illustrationManager = nil;
}
```

❶ Create a sub-manager for the very specific task of fetching thumbnail images to place in the table view cells. Provided that the coupling between `self.manager` and the view controller was set up properly (see previous example), if the user navigates away from the view controller and it is destroyed, the `self.manager` will be cancelled, which will cascade and also cancel this specific manager and any requests scheduled on it. Schedule all the image requests on the sub-manager.

❷ Schedule all the image requests on the sub-manager.

❸ If the user starts to scroll the table view, we cancel all the pending request by cancelling the sub-manager. This way network access will not impact on the scrolling animation.

5 Deezer model

In this section we present the model provided by Deezer's iOS SDK. You need to use these objects to query information about tracks, albums, artists, playlists, and users.

At first, you may feel that the model is somewhat odd and overly complex. The reality is that it is not exactly a model in the sense of glorified key-value stores. Behind the scenes, it manages and access remote objects presented by Deezer's web services. Fetching remote objects is an asynchronous task. Deezer's SDK manages this fetching for you, but this process is asynchronous.

The result is that most querying on objects are implemented as callbacks. This means you *might* entangle yourself in callback hell, but *should* be able to escape it easily. All the *eventual* network requests that must be scheduled to fulfill the information request will be handled by the SDK. The SDK will schedule these requests in the instance of the `DZRRequestManager` you provide (or the default manager if you do not give an instance).

The only issue concerning this interface is the stringly *stringly typed* aspect of it.

Should you specify a network manager?
As explained in the section on manager trees, it is good practice to tie the life of all possible issued network requests. This is in order to gather information to display in a view controller for this same view controller. The rule of thumb here is that if you request information from Deezer objects from a view controller, you should pass to a network manager owned by this view controller, and cancel this manager when you destroy the view controller.

Here is the interface of `DZRObject`. This is the base class of all model objects of the SDK and it defines most of the behaviour you will be using for these objects.

```
@interface DZRObject : NSObject <NSCopying>
@property (nonatomic, readonly) NSString* identifier;

+ (void)
   searchFor:(DZRSearchType)type withQuery:(NSString*)query          // ❶
   requestManager:(DZRRequestManager*)manager
   callback:(void(^)(DZRObjectList*, NSError*))callback;
+ (DZRNetworkRequest *)
  objectWithIdentifier:(NSString*)identifier                         // ❷
  requestManager:(DZRRequestmanager*)manager
  callback:(void(^)(DZRObject *o, NSError *error))callback;

- (NSArray*)supportedInfoKeys;                                       // ❸
- (NSArray*)supportedMethodKeys;                                     // ❹

- (id<DZRCancelable>)
  valueForKey:(NSString*)key                                         // ❺
  withRequestManager:(DZRRequestManager*)manager
  callback:(void(^)(id value, NSError *error))callback;
- (id<DZRCancelable>)
  valueForKeyPath:(NSString *)key                                    // ❻
  withRequestManager:(DZRRequestManager *)manager
  callback:(void (^)(id, NSError *))callback;
- (id<DZRCancelable>)
  valuesForKeyPaths:(NSArray*)keyPaths                               // ❼
  withRequestManager:(DZRRequestManager*)manager
  callback:(void(^)(NSDictionary*, NSError*))callback;

- (BOOL)isPlayable;                                                  // ❽
- (BOOL)isIllustratable;                                             // ❾
- (BOOL)isDeletable;                                                 // ❿
- (BOOL)isRatable;                                                   // ⓫
- (BOOL)isCommentable;                                               // ⓬
- (BOOL)isFlowable;                                                  // ⓭
- (BOOL)isRadioStreamable;                                           // ⓮
```

```
- (BOOL)isGhost;                                                    // ❶⑤

- (void)invalidateInfos;                                            // ❶⑥

+ (void)chartWithRequestManager:(DZRRequestManager*)manager         // ❶⑦
        numberOfItems:(NSInteger)numberOfItems
        callback:(void (^)(id object, NSError* error))callback;
@end
```

❶, ❷ Entry points. Use these messages to retrieve your first objects. After that, refrain from using these. Instead, query properties on the objects you have.

❸, ❹ These methods help you check which properties are available on each object. In the API documentation *InfoKeys* corresponds to the `INFO` section. These are properties that are intrinsic to the object. On the other hand, *MethodKeys*corresponds to the `METHODS` section. From a more technical point of view, `INFO` values will be automatically cached by the objects, whereas `MEHTODS` will not. Otherwise, the access to these two types of data is uniform and you can query them the same way.

❺, ❻, ❼ These 3 methods represent the query interface of `DZRObject`s from the more simple (`-[DZRObject valueForKey:withRequestManager:callback:]`) to the more complex and powerful(`-[DZRObject valuesForKeyPaths:withRequestManager:callback:]`). They will be detailed in the Querying information section.

❽, ❾, ❿, ⓫, ⓬, ⓭, ⓮ These 6 methods allow you to query the possibilities of the object. They correspond to the protocols explained in more detail later in a section below.

⓯ This method allows you to see if this particular object is a *ghost*. See the corresponding section on ghost objects for more information.

⓰ Finally this method allows you to clear the info cache for this object, forcing the system to again fetch the values from the network. Be careful when using this as it has performance implications.

⓱ This method allows you to retrieve top charts with number of items.

Important
The `-[DZRObject chartWithRequestManager:numberOfItems:callback:]` method, even if it is defined on `DZRObject` must be called on one of its subclasses like `DZRTrack`, `DZRAlbum`, `DZRArtist`, `DZRPlaylist` and **never** on the class `DZRObject` itself. This is because if you call directly on `DZRObject` the SDK is incapable to know which type of object you are requesting.

5.1 Model entry points

We mentioned that there are two methods you can use to initiate your interaction with the Deezer model.

5.1.1 Getting an object by type and identifier

The first(`[DZRObject objectWithIdentifier:requestManager:callback:]`)allows you to request an object if you know its identifier. Generally you use this method to request information from a user that just logged into their Deezer account via your application. Alternatively, your application perhaps stores identifiers of favourite songs or albums. You can retrieve the full information about these objects through a call to `-[DZRObject objectWithIdentifier:requestManager:callback:]`.

Important
The `-[DZRObject objectWithIdentifier:requestManager:callback:]` method, even if it is defined on `DZRObject` must be called on one of its subclasses like `DZRUser`, `DZRAlbum...` and **never** on the class `DZRObject` itself. This is because if you call directly on `DZRObject` the SDK is incapable to know which type of object you are requesting.

Example: Query information of the currently logged in user

```
- (IBAction)showUser:(id)sender {
    [DZRUser                                                    // ❶
     objectWithIdentifier:@"me"                                 // ❷
     requestManager:[DZRRequestManager defaultManager]          // ❸
     callback:^(DZRObject *o, NSError *error) {                 // ❹
         DeezerItemViewController* itemViewController = nil;
         itemViewController = [[DeezerItemViewController alloc]
                               initWithDZRObject:o];
         [[self navigationController]
           pushViewController:itemViewController animated:YES];
     }];
}
```

❶ We use the entry point `+[DZRObject objectWithIdentifier:requestManager:callback:]` on the `DZRUser` class, **not** on the base class `DZRObject` so that the system knows which type of object we are querying.

❷ `@"me"` is a special aliased identifier that corresponds to the current logged-in user. You do not have to know the real identifier of the user to access their information (although you could retrieve this information through the `DeezerConnect` object used to let the user log in).

❸ Here, for simplicity and because this is the API entry point, we use the default request manager even though this is a bad habit.

❹ We have our `DZRUser` bject again. Now we can use it. Here we give it to a dedicated view controller that will display the useful user information.

5.1.2 Searching for objects

The second entry point (`+[DZRObject searchFor:withQuery:requestManager:callback::]`) allows you to search for objects satisfying a query string. You can query several types of objects:

- Tracks
- Artists
- Albums
- Users

Example: Query for the tracks matching "sun"

```
[DZRObject                                                  // ❶
 searchFor:DZRSearchTypeTrack                               // ❷
 withQuery:@"sun"                                           // ❸
 requestManager:self.manager
 callback:^(DZRObjectList *results, NSError *error) {
          // ❹
}];
```

❶ This time we call the method directly on `DZRObject` as no type information is required.

❷ `DZRSearchTypeTrack` means that we are searching for track objects. You can also search for albums (`DZRSearchTypeAlbum`), artist (`DZRSearchTypeArtist`) or user (`DZRSearchTypeUser`). The search with auto-completion results is not yet implemented. If you need it, you will have to request the API directly.

❸ We are searching for matches on the query `@"sun"`.

❹ At this point, we have our object collection back (or an error). See the section corresponding to object collection for more details about `DZRObjectList`.

5.2 Querying information

Once you have got an object by using one of the calls described in the section about entry points, you now need to extract information or new objects from it. `DZRObjects` does not present you with a classical API for this - there is no property defined on the different subclass. Instead `DZRObjects` offers you an interface that mimics *Key Value Coding* but with the added complexity of asynchronism. This means that you will query information on objects through the use of keys (`NSStings`) or key paths (dotted notation for key chaining). As mentioned earlier, this is not ideal as it introduces possible error since the keys are not checked statically by the compiler (stringly typed). However, we felt this was a necessary evil as this interface would not be foreign to an Objectice-C programmer since it looks like *key Value Coding*.

5.2.1 Where to find the keys ?

The keys are the same as those presented in the documentation of the Deezer's web services' API. You should first look at this documentation. The second source of information is the couple of methods `-[DZRObject supportedInfoKeys]` and `-[DZRObject supportedMethodKeys]` which effectively list the supported keys for each object. The type of associated values can also be inferred based on the documentation of the Deezer API. The mapping is quite natural and there are no surprises except for the list of objects described in its own section.

API type	Objective-C type
strings	`NSString`
URL	`NSString`
boolean	`NSNumber`
int	`NSNumber`
objects	`DZRObject`
list	`DZRObjetList`
date	`NSDate`

5.2.2 Asynchronous KVC

As discussed above, `DZRObjects` re just proxies to remote objects stored on Deezer and accessible through the Deezer web service. Thus, retrieving information about `DZRObjects` may involve issuing network requests to the Deezer web service. That is why all the methods that allow querying information from `DZRObjects` have to be provided with a network manager and provide their result through a callback.

5.2.3 Querying the value associated with a key

The first and most simple way to query information from the model is by using the `-[DZRObject valueForKey:withRequestManager:callback:]` method. You pass a string key corresponding to the information you are seeking. The model will return the associated value (or an error) through the provided callback.

Example: fetching the track's title

```
[playingTrack
 valueForKey:@"title"                                  // ❶
 withRequestManager:self.manager
 callback:^(NSString *title, NSError *error)           // ❷
 {
    // ❸
 }];
```

❶ We request the title of the song passing the key `@"title"` to `-[DZRObject valueForKey:withRequestManager:callback:]`. We found this key in Deezer web service API documentation.

❷ In the API documentation it is specified that the `title` property is a string, so we know that we should get an `NSString` back in the callback. We choose to directly type the variable here instead of passing in an untyped pointer `id` and then checking the type. Depending on situation and confidence, the later approach may be more appropriate.

❸ The model, after having found the information in its cache or having fetched it from the network, returns it to you (or an error if something unexpected has arisen).

5.2.4 Querying information deeper in the object's graph

The method `-[DZRObject valueForKey:withRequestManager:callback:]` is good enough when you want just one piece of information directly about the object you hold. But if you need more distant information (for example even two hops away), it will be inconvenient as you will have to resort to using nested callbacks.

Example: Fetching artist's name from a track using nested callbacks (BAD)

```
[playingTrack
 valueForKey:@"artist"                              // ❶
 withRequestManager:self.manager
 callback:^(DZRArtist *artist, NSError *error)
 {
    // ❷
    [artist
     valueForKey:@"name"                            // ❸
     withRequestManager:self.manager
     callback:^(NSString *name, NSError *error)
     {
        // ❹
     }];
 }];
```

❶ First we request the artist of the track.

❷ We got the response from the model, here we should normally deal with eventual errors.

❸ If there was no error, it is time to request the name of the artist.

❹ We finally have the response we actually cared about and if no error is returned, we can now use the artist's name.

This example is simplified. Notably there is no error checking, no consideration for `weak` or `strong` pointers and retain cycles. Nevertheless, we already nested some callbacks and needed 8 lines of boilerplate. Here is a way to ask the model directly for what is of interest to us. The model then takes care of fetching intermediate objects and checking for errors (returning the actual error to you if needed). Enter `-[DZRObject valueForKeyPath:withRequestManager:callback:]` the cousin of `-[NSObjecy valueForKeyPath:]`.

Example: Fetching artist's name from a track using keypath (GOOD)

```
[playingTrack
 valueForKeyPath:@"artist.name"                     // ❶
 withRequestManager:self.manager
 callback:^(DZRArtist *artist, NSError *error)
 {
    // ❷
 }];
```

❶ We change the call from `-[DZRObject valueForKey:withRequestManager:callback]` to `-[DZRObject valueForKetPath:withRequestManager:callback:]` and pass in the same kind of keypath you would use for `-[NSObject valueForKeyPath:]`.

❷ The model did all the work for us and just returns the requested information (or an error).

5.2.5 Querying multiple keypaths in one call

Finally, you can query the model for a set of values through their keypath. The model will synchronise all the neccessary network queries and will fetch all the intermediate objects. The model will gather all necessary data in the background and will provide the result as a dictionary whose keys are the requested keypaths and values are the associated information.

Performance considerations
Since the model automatically fetches all the intermediate objects and internally eventually fetches objects from the network for each key of a keypath, you should be careful not to query objects that are too deep (with a long keypath). The number of properties you request (the `count` of the array of keys) on the other hand normally have little effect on the query's performance if you are querying *info*s but can be costly if you don't request*method*s.

Example: Querying multiple keypaths

```
[playingTrack                                                    // ❶
 valuesForKeyPaths:@["id", @"name", @"duration", @"artist.name"]  // ❷
 withRequestManager:[DZRRequestManager defaultManager]
 callback:^(NSDictionary *info, NSError *error) {
     player.trackID = info[@"id"];                               // ❸
     player.titleLabel.text = info[@"name"];                     // ❹
     player.trackDuration = [info[@"duration"] longValue];       // ❺
     player.artistLabel.text = info[@"artist.name"];             // ❻
 }];
```

❶ `playingTrack` is a `DZRTrack` object we got by other means and on which we want more information.

❷ We specify all the information that we want. We use a keypath here that can query deeper information following the links between objects. The system will take care to schedule the network requests properly.

❸, ❹, ❺, ❻ We get a dictionary back with the data we requested. The dictionary's keys are the same keypath you specified when you called the `-[DZRObject valuesForKeyPaths:withRequestManager:callback:]` method. This means that the keys have embedded dots in them. This means that the keys have embedded dots in them. You should be careful to then use `-NSDictionary`'s interface (indexed or `-[NSDictionary objectForKey:]`) instead of KVC to reduce the odds of confusion and the chance that calling `-[NSObject valueForKeyPath:]` that will have an unexpected effect.

5.3 Objects collection

Sometimes you will request a property that holds a collection of other `DZRObject`s. In this case, the model will return a special object instance of `DZRObjectList`. You can think of it as a wrapper around an `NSArray` with asynchronous access. It is bounded (has a `count` property) and you can query any element in the collections by its index. You are reminded that this is an asynchronous operation by the fact that you have to provide a `DZRRequestManager` and a callback: `-[DZRObjectList objectAtIndex:withManager:callback:]`.

If you know that you need more than just one object, you can ask for a range of objects instead: `-[DZRObjectList objectsAtIndexes:withManager:callback:]`. Finally, if you want to retrieve an array populated with all the elements of the collection, you can call the method `-[DZRObjectList allObjects:]`.

Performances considerations
Requesting a large range of objects on a `DZRObjectList` and a fortiori requesting all objects can be expensive in term of both of network traffic and processing time.

```
@interface DZRObjectList : NSObject
- (void)
  objectAtIndex:(NSUInteger)index
  withManager:(DZRRequestManager *)manager
  callback:(void (^)(id obj, NSError *error))callback;
- (void)
  objectsAtIndexes:(NSIndexSet *)indexes
  withManager:(DZRRequestManager *)manager
  callback:(void (^)(NSArray *objs, NSError *error))callback;
- (void)
  allObjectsWithManager:(DZRRequestManager *)manager
  callback:(void (^)(NSArray *objs, NSError *error))callback;
- (NSUInteger)count;
@end
```

5.4 Object's capabilities and protocols

The model provides some convenient protocols that inform you of the capabilities of objects. Generally these protocols are quite narrow and only expose a unique method. Think of these protocols as a note about what you can do with an object.

At runtime you can test the functionality of the objects using convenience methods defined on `DZRObject` that test the adherence of the objects class to the different protocols.

- `isPlayable`
- `isIllustratable`
- `isDeletable`
- `isRatable`
- `isCommentable`
- `isFlowable`
- `isRadioStreamable`

5.4.1 DZRPlayable

This protocol hints that this object can be passed to the `DZRPlayer` to be played. More details will be provided in the documentation of the player.

```
@protocol DZRPlayable <NSObject>
- (id<DZRPlayableIterator>)iterator;
@end
```

5.4.2 DZRIllustratable

Illustratable object are ones that have a visual representation. For example, an artist can be represented by a photo of this artist, and an album by its cover. This protocol offers a way to retrieve visual representation as an `UIImage` by the use of the method `-[DZRObject illustrationWithRequestManager:callback:]`. The callback's type is `void(^)(UIImage *, NSError \*)` where an `UIImage` will be passed back to you if the retrieval was successful.

```
@protocol DZRIllustratable <NSObject>
- (id <DZRCancelable>)
  illustrationWithRequestManager:(DZRRequestManager *)manager
  callback:(void(^)(UIImage *illustration, NSError *error))callback;
@end
```

For example, lets say that you have a table view with each cell presenting an album. You want to display the cover in the cells. When the underlaying scroll view has finished moving, you can request and display the covers of the visible cells in the delegate of your table view:

Example

```
- (void)scrollViewDidEndDecelerating:(UIScrollView *)scrollView
{
    self.illustrationManager = [self.manager groupingManger];                    // ❶
    [[objectsView visibleCells]
      enumerateObjectsUsingBlock:^(UITableViewCell *cell, NSUInteger idx, BOOL *stop)
    {
        DZRObject *o = [self.data
                        objectAtIndex:[objectsView indexPathForCell:cell].row];
        if (o.isIllustratable) {                                                 // ❷
            [(DZRObject<DZRIllustratable>*)o                                     // ❸
             illustrationWithRequestManager:self.illustrationManager
             callback:^(UIImage *illustration, NSError *error)
             {                                                                   // ❹
                 if (illustration) {
                     cell.imageView.image = illustration;
                     [cell setNeedsLayout];
                 }
             }];
        }
    }];
}
```

❶ Create a new sub-manager just for the purpose of fetching illustrations. This way, if the table view starts scrolling again, you can conveniently cancel all pending network requests.

❷ Just to make sure, test to see if the object is illustratable. This is a method on `DZRObject` that basically tests the implementation of the protocol by the object's class.

❸ Now request the illustration, passing our dedicated sub-manager.

❹ The SDK gives the illustration (or an error), setting it as the cell's image.

This example is simple and you should generally cache images

5.4.3 DZRCommentable

This protocol shows you that it is possible to comment on the object. You can post the comment by using the method `-[DZRObject postComment:withRequestManager:callback:]`. The comment is just text provided as an `NSString` and will be posted on behalf of the currently logged in Deezer user. The callback has the type `void(^)(DZRComment*, NSError*)` meaning that you will receive the object modelling the posted comment or an error.

```
@protocol DZRCommentable <NSObject>
- (void)postComment:(NSString*)text
 withRequestManager:(DZRRequestManager*)manager
           callback:(void(^)(DZRComment* comment, NSError *error))callback;
@end
```

5.4.4 DZRRatable

This protocol tells you that the user can assign the object a rating (from 1 to 5 stars). Again, only use the specified method in this protocol: `-[DZRObject rateObject:withRequestManager:callback:]` passing an `NSUInteger` between 1 and 5 included. between 1 and 5. The callback here can only report an eventual error and thus have the type `void(^)(NSError*)`.

```
@protocol DZRRatable <NSObject>
- (void)
  rateObject:(NSUInteger)note
  withRequestManager:(DZRRequestManager*)manager
  callback:(void(^)(NSError *error))callback;
@end
```

5.4.5 DZRDeletable

This protocol marks the object as being deletable.

```
@protocol DZRDeletable <NSObject>
- (void)
  deleteObjectWithRequestManager:(DZRRequestManager*)manager
  callback:(void(^)(NSError *error))callback;
@end
```

5.4.6 DZRFlowable

This protocol indicates that you can create a *Flow* radio from this object (`DZRUser`. Calling the method `-[DZRObject flowRadioWithRequestManager:managercallback:]` gives you a `DZRManagedRadio` which is a `DZRPlayable` object that you can directly hand to a `DZRPlayer` instance.

```
@protocol DZRFlowable <NSObject>
- (void)
  flowRadioWithRequestManager:(DZRRequestManager*)manager
  callback:(void(^)(DZRManagedRadio* radio, NSError *error))callback;
@end
```

Example: Play the user's flow radio

```
[DZRUser                                                          // ❶
 objectWithIndentifier:@"me"
 networkManager:[DZRRequestManager defaultManager]
 callback:^(DZRUser *me, NSError *error)
 {
    [me
     flowRadioWithRequestManager:[DZRRequestManager defaultManager] // ❷
     callback:^(DZRManagedRadio *radio, NSError *error)
     {
         [self.player play:radio];                                // ❸
     }];
 }];
```

❶ This example starts from nothing for demonstration purposes. In the real world, you will most likely already have a handle on the object you want to request the flow radio from.

❷ Now that we have a handle on our user, we can request its flow radio.

❸ We now have a flow radio which is a playable object what we can directly pass to a player.

5.4.7 DZRRadioStreamable

The protocol indicates that you can get a radio derived from this object, in the same way you can have a *Flow Radio* from a user.

```
@protocol DZRRadioStreamable <NSObject>
- (void)
  radioWithRequestManager:(DZRRequestManager *)manager
  callback:(void(^)(DZRManagedRadio *radio, NSError *error))callback;
@end
```

5.5 Ghost Objects

Sometimes the model will deliver what is considered to be a *ghost* object. Such objects have a subset of normal object information and no means to fetch more. If you try to get a property not defined on a ghost object, the model will inform you by delivering a `DZRModelUndefinedPropertyOnGhostObjectError` error code in the `DZRModelErrorDomain` error domain. This error is permanent and should be handled (for example by using a placeholder value). Trying to request this property again will result in the same error.

6 Deezer Player

The Deezer iOS SDK player is quite simple but should allow you to play everything you need. The player takes `DZRObjects` that are marked as playable (implements the `DZRPlayeble` protocols and thus respond `YES` to the `isPlayable` message). t plays the tracks contained in this playable object.

Currently, playable objects are :

- `DZRTracks`
- `DZRAlbums`
- `DZRPlaylists`
- `DZRRadios`
- `DZRManagedRadios`

All you have to do to play something is send the `-[DZRPlayer play:]` message to an initialised `DZRPlayer` instance.

```
@interface DZRPlayer : NSObject
@property (nonatomic, weak) id<DZRPlayerDelegate> delegate;
@property (nonatomic, assign) DZRPlayerNetworkType networkType;

@property (nonatomic, readonly) DZRTrack *currentTrack;
@property (nonatomic, readonly) size_t currentTrackDuration;

@property (nonatomic, assign) DZRPlaybackRepeatMode repeatMode;
@property (nonatomic, assign) BOOL shuffleMode;

- (id)initWithConnection:(DeezerConnect*)connection;

- (void)play:(id<DZRPlayable>)playable;
- (void)play:(id<DZRPlayable>)playable atIndex:(NSInteger)index;

- (void)pause;
- (void)play;
- (void)stop;
- (void)next;
- (void)updateRepeatMode:(DZRPlaybackRepeatMode)repeatMode;
- (void)toggleShuffleMode;

- (BOOL)isPlaying;
- (BOOL)isReady;
@end
```

```
@protocol DZRPlayerDelegate <NSObject>
@optional
- (void)player:(DZRPlayer*)player
     didBuffer:(long long)bufferedBytes outOf:(long long)totalBytes;
- (void)player:(DZRPlayer*)player
       didPlay:(long long)playedBytes outOf:(long long)totalBytes;
- (void)player:(DZRPlayer*)player didStartPlayingTrack:(DZRTrack*)track;
- (void)player:(DZRPlayer*)player didEncounterError:(NSError*)error;
- (void)playerDidPause:(DZRPlayer*)player;
@end
```

```
typedef NS_ENUM(NSInteger, DZRPlaybackRepeatMode)
{
    DZRPlaybackRepeatMode_None = 0,
    DZRPlaybackRepeatMode_AllTracks = 1,
    DZRPlaybackRepeatMode_CurrentTrack = 2
};
```

Use Repeat and Shuffle Mode
If your application need to repeat track or shuffle a tracklist, implementing the protocols `DZRPlayable` and `DZRPlayableRandomAccessIterator` is all you need to do for use this features with the `DZRPlayer`. To change mode, send the -[DZRPlayer updateRepeatMode:DZRPlaybackRepeatMode] or -[DZRPlayer toggleShuffleMode:] message to an initialised `DZRPlayer` instance.

6.1 DZRPlayable iterator

The only thing a `DZRPlayer`instance needs to play a collection of tracks is an iterator implementing the `DZRPlayableIterator` protocol.

```
@protocol DZRPlayableIterator
- (void)
  currentWithRequestManager:(DZRRequestManager *)requestManager
  callback:(DZRTrackFetchingCallback)callback;
- (void)
  nextWithRequestManager:(DZRRequestManager *)requestManager
  callback:(void(^)(DZRTrack *nextTrack, NSError *error))callback;
@end
```

This protocol is quite simple (this is an iterator after all). The player asks the iterator for the next `DZRTrack` to play in an asynchronous way. There are three possibilities depending on the response of the iterator:

- The player gets a `DZRTack`, it will play it.
- The player gets an error - it will pause and report the error to its delegate.
- If the iterator returns `nil` and no error, then the player considers the collection finished. At this point, sending control message (`-[DZRPlayer play]`, `-[DZRPlayer pause]`, ...) will have no effect. You will need to give the player a fresh iterator by sending a new `-[DZRPlayer play:]` message to continue using the player.

Create your own tracks collection
If your application required a customisable collection of tracks, implementing the protocols `DZRPlayable` and `DZRPlayableIterator` is all you need to do in order to make it readable by the `DZRPlayer`.

6.1.1 DZRPlayableRandomAccess iterator

This protocol is a subclass of `DZRPlayableIterator` with more asynchronous methods to handle the iterator.

```
@protocol DZRPlayableRandomAccessIterator <DZRPlayableIterator>
- (void)trackAtIndex:(NSUInteger)index
    withRequestManager:(DZRRequestManager *)manager
              callback:(DZRTrackFetchingCallback)callback;

- (void)previousWithRequestManager:(DZRRequestManager *)requestManager
    callback:(DZRTrackFetchingCallback)callback;

- (void)previousFrom:(DZRTrack *)track
    withRequestManager:(DZRRequestManager *)manager
              callback:(DZRTrackFetchingCallback)callback;
- (void)nextFrom:(DZRTrack *)track
    withRequestManager:(DZRRequestManager *)manager
              callback:(DZRTrackFetchingCallback)callback;

- (void)reset:(void(^)(NSError *error))callback;
- (void)count:(void(^)(NSUInteger count, NSError *error))callback;
- (NSInteger)currentIndex;
@end
```

This protocol working only with this playable objects :

- `DZRTrack`
- `DZRAlbum`
- `DZRPlaylist`

7 Migration examples

In this section we provide some examples of migration from the previous API to the new API.

7.1 Playing a track when the ID is known

Playing track (Previous API)

```
@interface ExampleVC () <DeezerRequestDelegate, PlayerDelegate, BufferDelegate>       // ❶
@property (nonatomic, strong) DeezerConnect *connect;                                 // ❷
@property (nonatomic, strong) PlayerFactory *player;                                  // ❸
@end

@implementation ExampleVC
- (id)init
{
    // ❹
    self = [super init];
    if (self) {
        self.player = [PlayerFactory createPlayerWithNetworkType:NetWork_WIFI_AND_3G
                                        andBufferProgressInterval:10.f];
        self.player.delegate = self;
        self.player.bufferDelegate = self;
    }
    return self;
}
```

```
- (void)playID:(NSString*)trackID                                          // ❺
{
    DeezerRequest *req = nil;
    req = [self.connect
      createRequestWithServicePath:[@"track" stringByAppendingPathComponent:trackID]  // ❻
      params:nil
      delegate:self];

    [self.connect launchAsyncRequest:req];                                 // ❼
}

- (void)playTrack:(NSDictionary*)track
{
    [self.player
        preparePlayerForTrackWithDeezerId:track[@"id"]                     // ❽
        stream:track[@"stream"]
        andDeezerConnect:self.connect];
    [self.player play];                                                    // ❾
}

#pragma mark DeezerRequestDelegate

- (void)request:(DeezerRequest *)request didReceiveResponse:(NSData *)response        // ❿
{
    NSError *err = nil;
    NSDictionary *track = [NSJSONSerialization
    JSONObjectWithData:response options:nil error:&err];

    if (err) return;
    else {
        [self playTrack:track];
    }
}
@end
```

❶ Declaration of your view controller. In this example we just show implementation of parts of these protocols (`DeezerRequestDelegate`).

❷, ❸ You need a `PlayerFactory` and connection information to identify the logged-in user.

❹ Some initialisations. Here just the player.

❺ The journey really starts here. You are passing the `trackID`.

❻ You need to configure a network request, knowing the web service path (and constructing it).

❼ Next, launch the request.

❿ The request notifies the view controller that it received a response from the network. Parse the JSON and pass the result to the next part to actually play the track.

❽ Configure the player.

❾ OK, let's play.

Playing track (New API)

```
@interface ExampleVC <DZRPlayerDelegate>                          // ❶
@property (nonatomic, strong) DZRPlayer *player;                  // ❷
@property (nonatomic, strong) DZRRequestManager *manager;         // ❸
@property (nonatomic, strong) id<DZRCancelable> trackRequest      // ❹
@end
```

```
@implementation
- (void)initWithConnection:(DeezerConnect*)connect                          // ❺
{
    self = [super init];
    if (self) {
        self.player = [[DZRPlayer alloc] initWithConnection:connect];
        self.player.networkType = DZRPlayerNetworkTypeWIFIAnd3G;
        self.manager = [[DZRRequestManager defaultManager] subManager];
    }
    return self;
}
- (void)playTrackID:(NSString*)trackID                                      // ❻
{
    [self.trackRequest cancel];                                             // ❼
    [self.player stop];                                                     // ❽

    self.trackRequest = [DZRTrack                                           // ❾
        objectWithIdentifier:trackID
        requestManager:self.manager
        callback:^(DZRTrack *track, NSError *error) {
            [self.player play:track];                                       // ❿
    }];
}
@end
```

❶ Declaration of your view controller. You just need to implement the `DZRPlayer` protocol. It is not effectively implemented in this example.

❷, ❸, ❹ You need a player. You will also need your personal network manager and a cancelable, explained later.

❺ Initialisation. You create your player and specify the network it can buffer on. Like we explained earlier, you should use a network manager.

❻ The real story begins (and ends) here again.

❼, ❽ First, make sure to stop the playback of any tracks previously started. Here we cancel the (hypothetical) network query that may still be running.

❾ We request the track with the given identifier.

❿ When the model returns the track, we give it to the player to play.

7.2 Playing a radio without login

You may also want to play a radio without having to make a user log in to a Deezer account.

Play a radio without user connected (Old API)

```
@interface DZRXViewController () <DeezerSessionDelegate,        // ❶
                                  DeezerRequestDelegate,        // ❷
                                  PlayerDelegate>               // ❸
{
}
@property (nonatomic, strong) DeezerConnect *connect;           // ❹
@property (nonatomic, strong) DeezerRequest *radioRequest;      // ❺
@property (nonatomic, strong) PlayerFactory *player;            // ❻
@end

@implementation DZRXViewController
```

```
- (id)initWithNibName:(NSString *)nibNameOrNil
               bundle:(NSBundle *)nibBundleOrNil
{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
    if (self) {
        self.connect = [[DeezerConnect alloc]                        // ❼
                        initWithAppId:@"139535" andDelegate:self];
        self.player = [PlayerFactory createPlayer];                  // ❽
        self.player.playerDelegate = self;                           // ❾
    }
    return self;
}

#pragma mark

- (void)fetchTrack
{
    self.radioRequest = [self.connect                            // ❿
     createRequestWithServicePath:@"radio/34275/tracks?current=true"
     params:nil
     delegate:self];
    [self.connect launchAsyncRequest:self.radioRequest];     // ⓫
}

- (IBAction)play:(id)sender
{
    [self fetchTrack];                                           // ⓬
}

#pragma mark DeezerRequestDelegate

- (void)request:(DeezerRequest *)request didFailWithError:(NSError *)error
{

}

- (void)request:(DeezerRequest *)request didReceiveResponse:(NSData *)response
{
    NSDictionary * track = [NSJSONSerialization                  // ⓭
                            JSONObjectWithData:response
                            options:0
                            error:nil];
    [self.player                                                 // ⓮
     preparePlayerForTrackWithDeezerId:track[@"id"]
     stream:track[@"stream"]
     andDeezerConnect:self.connect];
    [self.player play];                                          // ⓯
}

#pragma mark PlayerDelegate

- (void)player:(PlayerFactory *)player didFailWithError:(NSError *)error
{

}

- (void)player:(PlayerFactory *)player stateChanged:(DeezerPlayerState)playerState
{
    if (playerState == DeezerPlayerState_Finished ||
        playerState == DeezerPlayerState_Stopped) {
        [self fetchTrack];                                       // ⓰
    }
```

```
}

- (void)player:(PlayerFactory *)player timeChanged:(long)time
{

}

#pragma mark DeezerSessionDelegate

- (void)deezerDidLogin                          // ⓱
{
}

- (void)deezerDidLogout                         // ⓲
{
}

- (void)deezerDidNotLogin:(BOOL)cancelled       // ⓳
{
}
@end
```

❶, ❷, ❸ You first need to declare some protocols: `DeezerSessionDelegate` because you need it to initialise the `DeezerConnect` object, `DeezerRequestDelegate` because you will request the tracks from the radio, and finally `PlayerDelegate` because you need to know the status of the player to be able to fetch a new track when the current one ends.

❹, ❺, ❻ You will need a `DeezerConnect` object, you will also need a `DeezerRequest` to request the tracks. And obviously you need a `PlayerFactory` object to play the radio tracks.

❼ You need to initialise your `DeezerConnect` object with you application identifier, setting your view controller as the delegate.

❽, ❾ Next, initialise a `PlayerFactory` and set your view controller as the delegate.

❿ To request the current track from a radio, you need its identifier, ask for its tracks and specify with a query parameter that you only want the current one that is currently playing.

⓫ Then you launch the request.

⓬ When your user plays the radio, start by fetching the radio's current track info.

⓭ When the request come returns a response, you need to parse it.

⓮, ⓯ Then configure the player to play this track and tell the player to start playing.

⓰ When the player finishes playing the current track or when a playing error occurs, ask for the next track and the process will start again.

⓱, ⓲, ⓳ The implementation of the `DeezerSessionDelegate` protocol is empty since you will not log in any users.

As you can see, the playing of a radio requires a lot of boilerplate, even though this example is simplified and does not deal with any errors. Furthermore, the management of track chaining is really not ideal here as we start buffering the next track only when the current track ends. If the network is not stable enough, the user will experience quite a long gap between each track. In a real implementation, you should keep two players initialised and start the buffering of the next track before the end of the current one.

Now let's see how you can play the same radio with the new API.

Play a radio without user connected (New API)

```
@interface DZRXViewController ()
@property (nonatomic, strong) DeezerConnect *connect;          // ❶
@property (nonatomic, strong) DZRPlayer *player;                // ❷
@property (nonatomic, strong) DZRRequestManager *manager;      // ❸
@end

@implementation DZRXViewController
- (id)initWithNibName:(NSString *)nibNameOrNil
               bundle:(NSBundle *)nibBundleOrNil
{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
    if (self) {
        self.connect = [[DeezerConnect alloc]                          // ❹
                        initWithAppId:@"139535" andDelegate:self];
        self.player = [[DZRPlayer alloc]                               // ❺
                       initWithConnection:self.connect];
        [DZRRequestManager                                             // ❻
         defaultManager].dzrConnect = self.connect
        self.manager = [[DZRRequestManager defaultManager]             // ❼
                        subManager];
    }
    return self;
}

- (IBAction)play:(id)sender
{
    [DZRRadio                                                  // ❽
     objectWithIdentifier:@"139535"
     requestManager:self.manager
     callback:^(DZRObject *o, NSError *err) {
       [self.player play:o];                                   // ❾
    }];
}
@end
```

❶, ❷, ❸ In this version, you also need a `DeezerConnect` instance, together with a player and a network manager.

❹ You start off by initialising the SDK by instantiating the `DeezerConnect` instance with your application identifier and setting its delegate.

❺ Next, you can instantiate your `DZRPlayer` instance giving it your connect.

❻, ❼ Finally you configure your network stack with the Deezer connect and instantiate a sub-manager for your own use.

❽ Time to play the radio. Request your radio object by its identifier.

❾ The model returns your `DZRRadio` object, just give it the player.

This version is a little shorter than the previous one. The error management is still missing and some other minor concerns from the block approach are also missing. But more importantly, the goal is clearer in this version as there is less boilerplate to obscure what you want to do.

8 Compatibility layer

In order to ease the transition to the new version of the Deezer starting at version 0.9.0-beta4 we included a compatibility layer for the network requests. We did not include a compatibility layer for the player as we felt that the capabilities of the previous implementation were too limited anyway.

To use the compatibility layer, you have to include the `Deezer+Compatibility.h` header.

8.1 Network requests creation

Compatibility layer for DeezerConnect

```
@interface DeezerConnect (Compatibility)
- (DeezerRequest*)createRequestWithServicePath:(NSString*)servicePath
                                        params:(NSDictionary*)params
                                    httpMethod:(HttpMethod)httpMethod
                                responseFormat:(ResponseFormat)responseFormat
                                      delegate:(id<DeezerRequestDelegate>)delegate;
- (DeezerRequest*)createRequestWithServicePath:(NSString*)servicePath
                                        params:(NSDictionary*)params
                                    httpMethod:(HttpMethod)httpMethod
                                      delegate:(id<DeezerRequestDelegate>)delegate;
- (DeezerRequest*)createRequestWithServicePath:(NSString*)servicePath
                                        params:(NSDictionary*)params
                                      delegate:(id<DeezerRequestDelegate>)delegate;
- (DeezerRequest*)createRequestWithUrl:(NSString*)url
                                params:(NSDictionary*)params
                            httpMethod:(HttpMethod)httpMethod
                        responseFormat:(ResponseFormat)responseFormat
                              delegate:(id<DeezerRequestDelegate>)delegate;
- (DeezerRequest*)createRequestWithUrl:(NSString*)url
                                params:(NSDictionary*)params
                            httpMethod:(HttpMethod)httpMethod
                              delegate:(id<DeezerRequestDelegate>)delegate;
- (DeezerRequest*)createRequestWithUrl:(NSString*)url
                                params:(NSDictionary*)params
                              delegate:(id<DeezerRequestDelegate>)delegate;
- (void)launchSyncRequest:(DeezerRequest*)request;
- (void)launchAsyncRequest:(DeezerRequest*)request;
- (void)cancel:(DeezerRequest*)request;
@end
```

As you can see, all the requests constructing functions that were previously present on the DeezerConnect class are now part of this category. Some restrictions are present though:

- Response format are limited to JSON and XML. Asking for JSONP of PHP will rise an exception.
- The delegate method `requestDidStartLoading:` will never be called. The hook is not implemented yet.

8.2 Create DZRObjects from response

In order to interact with the player, you will need to transform the NSData returned to your request delegate to some DZRObject.

DZRObject compatibility layer

```
@interface DZRObject (Compatibility)
+ (DZRObject*)objectFromJSONData:(NSData*)JSONData error:(NSError**)error;
@end
```

Provided you requested JSON response format (this is the default), you can give the `NSData` directly to the `+[DZRObject objectFromjsondata:error:]` method. This method will handle for you the error responses from the server and give you a full fledged `DZRObject` of the correct type. If this object is a playable instance, you can give it directly to the DZRPlayer.