iOS 7 by tutorials

By the raywenderlich.com Tutorial Team & friends:
Christine Abernathy, Soheil Moayedi Azarpour, Colin Eberhardt, Charlie Fulton, Matt Galloway, Greg Heo, Matthijs Hollemans, Felipe Laso Marsetti, Jeremy Olson, Pietro Rea, Marin Todorov, Cesare Rocchi, Jamie Syke, Chris Wagner

Bonus Chapters

Chapter 25: Beginning Inter-App Audio

By Matt Galloway

Core Audio has been part of iOS since the very first public SDK; it underpins the entire audio system on iOS. It's quite a complex creature that, fortunately, most developers don't need to touch. The API is built in C, rather than Objective-C, and many developers find it quite tricky to come to grips with.

In iOS 7, Apple added inter-app communication to Core Audio, allowing you to send audio between apps. Now you can write an instrument app that sends audio to a recording app, such as GarageBand. Or you could write an app that takes audio from other apps, processes it and then returns it. The possibilities are endless with this technology.

The astute reader may realize that this technology sounds familiar. If so, it's because it's already been implemented in an app called "Audiobus" - http://audiob.us. This is a third party app that makes it possible to record audio in one app, and modify or mix it in another. Core Audio's inter-app audio is very similar in function, but it offers greater performance since it's deeply integrated into the operating system.

In this chapter, you'll work on two projects. The first is a guitar synthesizer and the second is an effects box. You'll take these starter projects and add inter-app audio capability to connect the two apps so that your synthesized guitar playing will be routed through the effects app.

This chapter doesn't cover the finer details of Core Audio; however, the next chapter explains how the Core Audio parts of the two apps work together. You may wish to read the next chapter immediately after finishing this chapter, or simply dip into it as you're reading this one.

Getting started

To get you up and running quickly with this tutorial you'll find two starter projects in the resources for this chapter: **iGuitar** and **iEffects**. Open both of the starter projects in Xcode and have a quick look.

iGuitar is a simple instrument app. Run it and play around with the guitar simulator. If you're a guitar player then you probably already know how play a chord, otherwise, select a chord from the bottom of the screen and then strum the strings by dragging your finger over them.

iEffects is an effects box app. It's set up to provide reverb, echoing the input sound as it might be heard in a large reflective room. Running this app will echo the sound picked up by the microphone through the speakers after passing it through the reverb effect. For more information about the reverb effect, check out the Wikipedia article http://en.wikipedia.org/wiki/Reverberation.

> **Note:** The effects app loops back audio from the microphone when not used in inter-app audio mode. Therefore to avoid a feedback loop (where the sound builds into a deafening blur!) you should plug in some headphones.

Once you've had a look around the starter projects, you're ready to move on to the next part of the tutorial.

First up are some background details on inter-app audio that you'll need before moving onto integrating this feature into the sample apps.

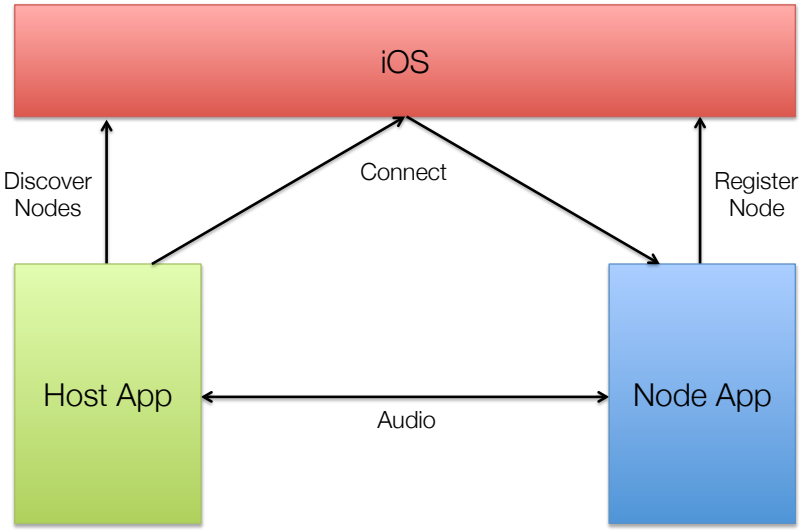
Basics of Inter-App Audio

The audio subsystem of iOS controls the inter-app audio and handles all of the hardware interfaces, ensuring that apps get access to the hardware as they need it. It also interrupts the audio stream of an app when required, such as when taking a phone call.

In order to participate in inter-app audio, your app must register with the audio subsystem. You can do this through the **Info.plist** file, similar to how you would set the background mode or other app characteristics.

Once this is done, registered apps can talk to each other by requesting a connection to one another. Within this connection, one app always acts as the host app and one as the node app. A host can talk to multiple nodes, but a node can have only one host. Once connected, audio can flow between the apps.

iOS provides a discovery method that returns all available nodes so a host knows which nodes it can communicate with. The discovery, connection and audio flow is demonstrated in the image below:



In fact, host apps and node apps can communicate more data than just audio data. They can communicate MIDI (**Musical Instrument Digital Interface**) events to provide a playback interface. For example, an app could show a keyboard interface and pass the notes played to an app that makes cat meows at different pitches.

Apps can also perform remote control actions. For example, nodes can tell their host to start and stop both recording and playback of audio. Similarly, hosts can communicate their recording and playback state to their node or nodes.

Once an app has registered with inter-app audio, it publishes what are known as **audio units**; it is through these that audio flows between apps. There are four types of audio units, each with varying inputs and outputs:

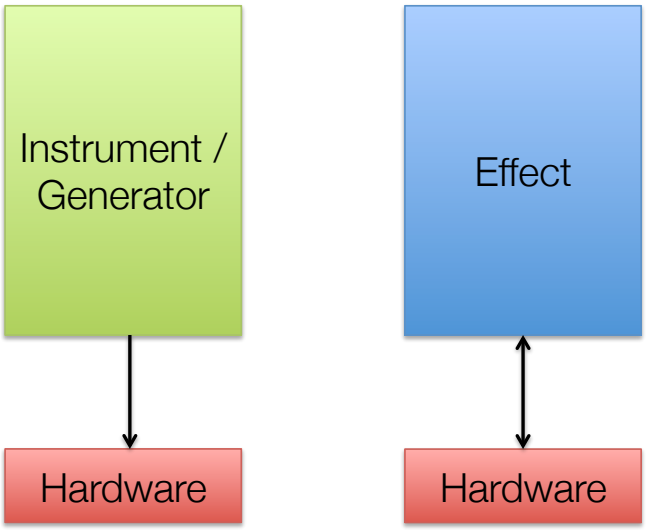
Category	Inputs	Outputs
Generator	*None*	Audio
Instrument	MIDI	Audio
Effect	Audio	Audio
Music Effect	Audio & MIDI	Audio

Note that an app doesn't necessarily have to publish a node to participate in inter-app audio. In fact, the host app often doesn't publish a node because its job is simply to coordinate audio. An example of this would be an app whose job it is to mix and record audio. It doesn't actually create any audio or add effects, but simply mediates audio created or processed by other apps.

The simplest combination of inter-app audio apps is one app that creates audio and passes the audio data to another app that adds an effect to that audio — which is exactly what the sample apps do in this chapter.

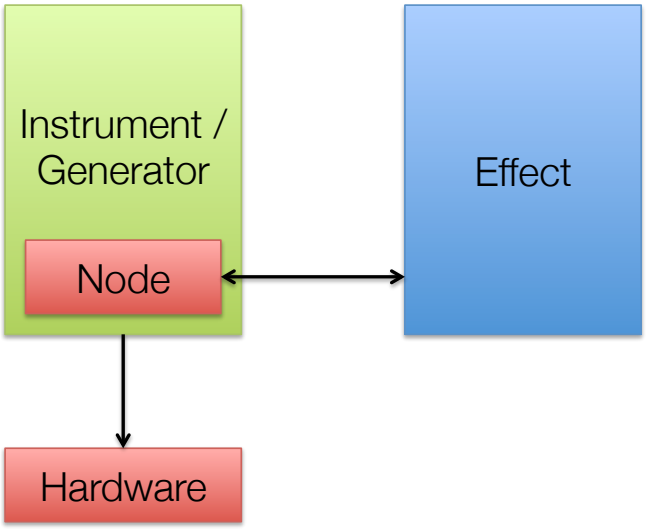
The guitar app can be thought of as either a generator or an instrument. It could either accept MIDI input to control the note playback, or alternatively allow the user to strum the guitar in the app to produce audio. On the other hand, the effects box app is purely an effect audio unit.

The diagram below shows how both apps would talk directly to the hardware in the absence of inter-app audio:



Notice that the effects app takes input and sends output to the hardware, whereas the guitar app only sends output. Once inter-app audio is introduced to the apps, however, the flow of audio data changes slightly.

Once the effect app publishes itself as having an effect node, the guitar app can request access to that node. Then, instead of playing the guitar audio directly to the hardware, the guitar app routes the audio through the effect node, which then processes the audio and sends it back to the guitar app. The guitar app then outputs the processed audio to the hardware. The diagram below illustrates the change in this flow:



The interesting thing about inter-app audio is that the effect app has no knowledge that anything has changed in the new scenario. Thanks to the API layer, it actually still thinks it is talking to the hardware directly. But the audio subsystem of iOS changes the routing for the effect app so that the input from hardware is now the output of the guitar app, and the output to hardware is now the input to the guitar app.

Now that you've got some background on inter-app audio, it's time to start connecting the two projects.

Publishing an audio unit

The first thing you're going to do is to make the effects app publish an audio unit with inter-app audio. There are a few steps you need to do for this to work. They are as follows:

1. Enable inter-app audio by adding an app entitlement.
2. Add inter-app audio to your provisioning profile.
3. Turn on the audio background mode, since inter-app audio apps need to be able to use audio when they are not the foreground application.
4. Add a key to the **Info.plist** file to tell iOS what type of audio unit the app has to share.
5. Write code to publish the actual audio unit when the app starts.

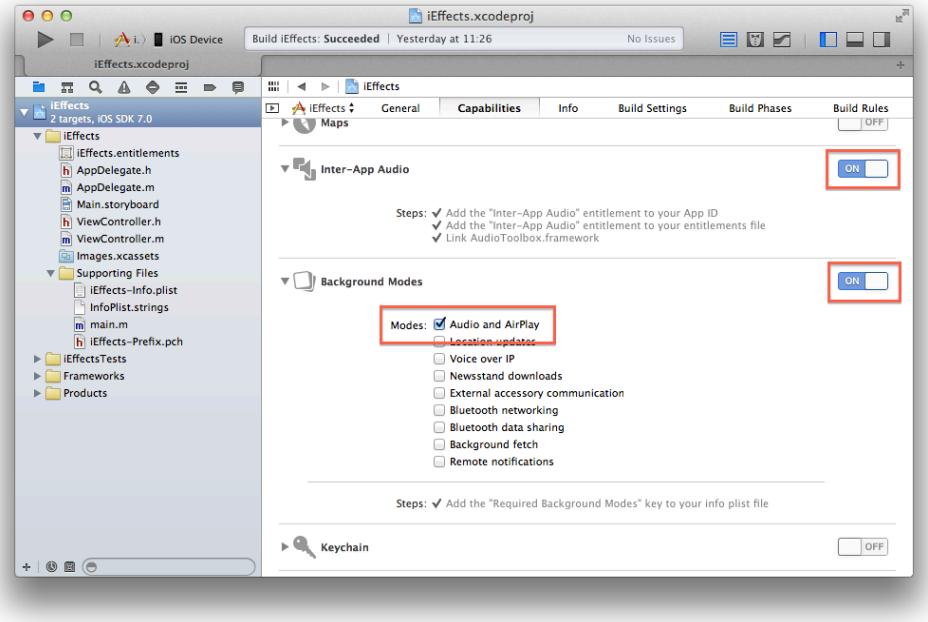
Fortunately, Apple has made the first three steps extremely easy to do in Xcode 5 thanks to the new Capabilities tab.

Open **iEffects** and select the project at the top of the project navigator. Then select the **Capabilities** tab. Scroll down and select the **Inter-app Audio** option; it describes what it will do once you turn it on. In a nutshell, Xcode will go off to the

provisioning portal, do what it needs to with profiles, and then magically do everything for you.

Turn on **Inter-app Audio** and sit back and watch Xcode do everything for you. If you're part of more than one iOS developer program, it will ask you which program you want to use. Select the one that makes sense for your current context and click **Choose**.

Next, turn on **Background Modes** and tick the **Audio and AirPlay** box. Your screen should now look like the following:



You'll notice that a new file named **iEffects.entitlements** has appeared in the project navigator. This file adds a key to the app so that iOS knows this app uses inter-app audio when code-signing the application.

That takes care of the first three steps in publishing an audio unit. Next up is adding a key to **Info.plist** to indicate what type of audio unit this app publishes. iOS needs to know what audio units will be published before the app is run; having that key in the plist means iOS can look up that information right away.

You can edit the **Info.plist** file directly in the project editor, but adding values to the file gets a bit fiddly sometimes. For the sake of this tutorial, you'll edit it directly.

Open up the **Supporting Files** folder in the project navigator, right-click on **iEffects-Info.plist** and select **Open As\Source Code**. The file will open in the main tab as the raw property list source.

Scroll to the bottom of the file and add the following before the `</dict>` line:

```
<key>AudioComponents</key>
<array>
    <dict>
        <key>manufacturer</key>
        <string>i7bt</string>
        <key>name</key>
        <string>iEffects</string>
        <key>type</key>
        <string>aurx</string>
        <key>subtype</key>
        <string>iasp</string>
        <key>version</key>
        <integer>1</integer>
    </dict>
</array>
```

This defines the audio units available in this app; it's stored as an array because an app could publish more than one unit. For example, your app could be both an effect and an instrument. The table below shows the possible keys in each dictionary that could describe a unit:

Key	Type	Description
Manufacturer	String	A four-character code representing the manufacturer of this audio unit. For this tutorial you've used '`i7bt`'. Apple uses '`aapl`'. This is used internally within Core Audio as a way to distinguish between different audio unit manufacturers. You should choose something that makes sense to your app. It is not detrimental to clash with another manufacturer, but you should strive to choose a unique code.
Name	String	The name of the unit. 'iEffects' makes sense in this case. You should choose a name that makes sense for your app. Other apps may use this to display to the user which units are available.
Type	String	A four-character code to define the type of audio unit. This is a remote effect so '`aurx`' is used. See below for

		details about other values.
Subtype	String	A four-character code to define the subtype. This must be '`iasp`' because that indicates an inter-app audio unit.
version	Integer	The version. You'll generally use '1' for this. For inter-app audio, the version doesn't really matter. This is present for other custom audio units (only available on Mac OS X anyway) where versioning matters because each may have a different API.

The values to change in your own app are the **manufacturer**, **name** and **type**. The type can be one of four values, representing each of the four kinds of inter-app audio units:

Audio unit type	Type four character code
Generator	`aurg`
Instrument	`auri`
Effect	`aurx`
Music Effect	`aurm`

Okay, you're almost there; you now need to publish the audio unit when the app starts.

Open **ViewController.m** and add the following method:

```
- (void)publishAsNode {
    AudioComponentDescription desc = {
        kAudioUnitType_RemoteEffect,
        'iasp',
        'i7bt',
        0,
        1
    };
AudioOutputUnitPublish(&desc, CFSTR("iEffects"), 0, _ioUnit);
}
```

The values here may look familiar; that's because you just added them to **Info.plist**. The values of the published node must match those defined in the property list. The code above registers with iOS that the app's `_ioUnit` audio unit should be published as a potential inter-app audio unit; once this is done, other apps may request access to that audio unit.

> **Note:** If you aren't familiar with Core Audio's audio units, you may want to read the "Remote IO Audio Unit"section in the following chapter.

The iEffects app's `_ioUnit` is a remote IO unit, meaning it accesses the hardware. Recall that when inter-app audio is connected, the audio subsystem reroutes audio such that the effect still talks to its hardware unit, but the "hardware" is now the host app.

Finally, add the following line to `viewDidLoad`, immediately after the call to `createAUGraph`:

```
[self publishAsNode];
```

This ensures that the audio unit is published once the audio graph has been created.

Build and run your app; nothing visible has changed at this point. (Sorry!) However, this ensures that your app still compiles and runs. The next section shows how to modify your iEffects app to ensure it responds when an app is connected.

Detecting a connected host app

Currently the audio graph is only started when the app is in the foreground, but you also need it to start when another app is connected. Without that, audio won't be processed by the app and inter-app audio won't work; the host app would think it was passing audio, but it would only get silence in return.

To detect the connection of the host app, you need to listen for state changes on the audio unit. This is similar to listening for notifications with `NSNotificationCenter`, where you register an observer for a particular notification.

Open **ViewController.m** and find the class continuation category at the top. Add the following method prototype:

```
- (void)audioUnitPropertyChanged:(void *)object
                            unit:(AudioUnit)unit
                      propID:(AudioUnitPropertyID)propID
                       scope:(AudioUnitScope)scope
                     element:(AudioUnitElement)element;
```

This is the method that's called back on the view controller, but since audio units are a C API, they require a C function as the callback.

Add the following function after the class continuation category but before the implementation block:

```
void AudioUnitPropertyChanged(void *inRefCon,
                              AudioUnit inUnit,
                              AudioUnitPropertyID inID,
                              AudioUnitScope inScope,
                              AudioUnitElement inElement)
{
    ViewController *SELF = (__bridge ViewController *)inRefCon;
    [SELF audioUnitPropertyChanged:inRefCon
                              unit:inUnit
                            propID:inID
                             scope:inScope
                           element:inElement];
}
```

This C function takes an opaque pointer: `inRefCon`. When you register the callback, you pass the view controller `self` pointer as the opaque pointer. In the above function you can cast the pointer to a `ViewController` pointer and then call the Objective-C method. It feels a bit roundabout, but this kind of C to Objective-C dance is common when dealing with C APIs.

Now, add the following method to the `ViewController` class:

```
- (void)audioUnitPropertyChanged:(void *)object
                            unit:(AudioUnit)unit
                          propID:(AudioUnitPropertyID)propID
                           scope:(AudioUnitScope)scope
                         element:(AudioUnitElement)element
{
    if (propID == kAudioUnitProperty_IsInterAppConnected) {
        UInt32 connected;
        UInt32 dataSize = sizeof(UInt32);
        AudioUnitGetProperty(_ioUnit,
                             kAudioUnitProperty_IsInterAppConnected,
                                 kAudioUnitScope_Global,
                                 0,
                                 &connected,
                                 &dataSize);

        _connected = (BOOL)connected;

        [self startStopGraphAsRequired];
```

```
    }
}
```

This method looks at the inter-app audio connection state of the remote IO audio unit; remember, that's the audio unit that was published as inter-app audio enabled. It then sets an instance variable and calls the method to start or stop the graph as required. The graph is only started if the app is in the foreground or if inter-app audio is connected.

Finally, to make all this work, you need to register the callback you just wrote. Find `createAUGraph` and replace the following comment:

```
/* TODO: REGISTER PROPERTY LISTENER */
```

...with the code below:

```
AudioUnitAddPropertyListener(_ioUnit,
                             kAudioUnitProperty_IsInterAppConnected,
                                 AudioUnitPropertyChanged,
                                 (__bridge void*)self);
```

This adds a listener for the "is inter-app connected" property of the audio unit. Recall that the opaque pointer expected by the callback is the view controller. That's why `self` is passed to this method.

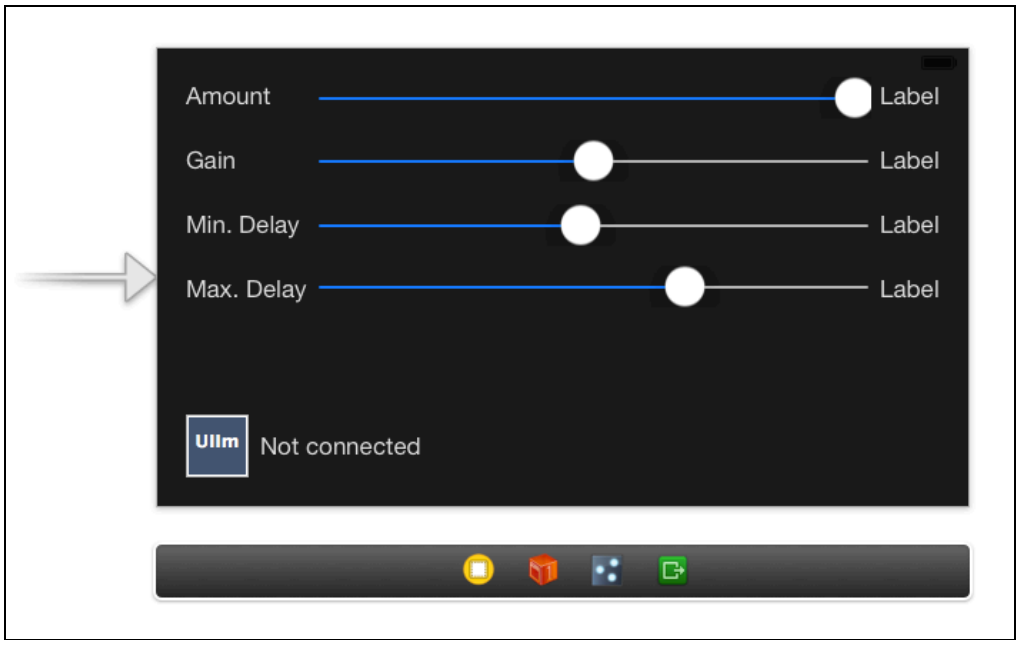
The app will now start its audio graph as required when inter-app audio is connected.

Adding UI to show connection state

It's all well and good that the app starts its audio graph as required, but it would be nice to give some visual feedback as to the state of the connection. You're now going to add an image view and a label to show this state.

Inter-app audio is quite clever and can even provide you with the icon of the connected app. The image view can then display the app icon and a label to indicate the state: connected or not connected.

Open **Main.storyboard** and find the view controller's view. Add a 44x44 image view to the bottom left hand corner of the screen. Then add a label next to it with the default text of “Not connected”. Your scene should now look like this:



Now connect these up to outlets in the view controller class called `connectedAppIcon` for the image view and `connectionStateLabel` for the label.

Next, add the following method to **ViewController.m**:

```
- (void)updateConnectedAppViews {
    if (_connected) {
        _connectionStateLabel.text = @"Connected";
        _connectedAppIcon.image =
            AudioOutputUnitGetHostIcon(_ioUnit, 44.0f);
    } else {
        _connectionStateLabel.text = @"Not connected";
        _connectedAppIcon.image = nil;
    }
}
```

This sets up the label and image as required. Notice the interesting bit that calls `AudioOutputUnitGetHostIcon`. This is the part of the new API for inter-app audio that allows you to obtain the app icon for the host app as previously discussed.

Finally, find `audioUnitPropertyChanged:unit:propID:scope:element:` and add the following line after the call to `startStopGraphAsRequired`:

```
[self updateConnectedAppViews];
```

Now, when the connection state changes, the views will update as required.

Build and run the app to ensure everything still works. The connection label at present, sadly, will display "Not connected" as there is nothing yet to connect to!

Opening the host app

To make the effects app even better, you can add a feature that will open the host app when its icon is tapped. As part of inter-app audio, Apple has made it possible to obtain a one-shot URL to open the connected app. This is very neat because iOS provides no direct mechanism for opening another app, other than knowing its URL scheme — that is, if it even *has* a URL scheme.

To add the capability of opening the host app, you're going to use a tap gesture recognizer on the image view you just added. Open **ViewController.m** and add the following method:

```
- (IBAction)openHostApp:(id)sender {
    if (_connected) {
        CFURLRef url;
        UInt32 size = sizeof(url);
        OSStatus result =
            AudioUnitGetProperty(_ioUnit,
                                 kAudioUnitProperty_PeerURL,
                                 kAudioUnitScope_Global,
                                 0, &url, &size);
        if (result == noErr) {
            [[UIApplication sharedApplication]
                openURL:(__bridge NSURL*)url];
        }
    }
}
```

If the app is connected to a host, then this method grabs the one-shot URL by requesting a special property of the audio unit. This uses the CoreFoundation `CFURLRef` type, which is handily toll-free bridged to `NSURL` such that it can be passed to the `openURL:` method of `UIApplication`.

Next, open **Main.storyboard** and select the host app icon image view. Turn on "**User Interaction Enabled**" under the attributes inspector. Next, drag a tap gesture recognizer onto the app icon. Then find the gesture recognizer in the list of objects on the left and wire it up to the `openHostApp:` method on the view controller.

Congratulations — the work on iEffects is done! Build and run the app to ensure everything still works. Next you're going to add the inter-app communication so that the guitar app can talk to the effects app. This is where the fun really begins!

Plugging in the guitar

The effects app is now ready for the guitar app to be (virtually) plugged in. In this part of the tutorial you're going to add the ability to route the audio from the guitar through the effects app — or any other inter-app audio you happen to have for that matter!

Open the **iGuitar** app and go to the project capabilities tab in the project settings. Once again, turn on **Inter-App Audio** and **Background Modes**, selecting **Audio & AirPlay**.

The next thing to do is provide a mechanism for picking which effect to connect. The best way to do this is through a table view controller that provides a list of the published audio units.

Remember how iEffects registered as an inter-app audio node in the plist and in `publishAsNode`? Now you can get a list of all published audio units from Core Audio. The first thing you'll need is a model class to store this data.

Add a class called **Effect** to the project, making it a subclass of **NSObject**. Then open **Effect.h** and change its contents to the following:

```
#import <Foundation/Foundation.h>

@import AudioUnit.AudioComponent;

@interface Effect : NSObject

@property (nonatomic, strong) NSString *name;
@property (nonatomic, assign)
    AudioComponentDescription componentDescription;
@property (nonatomic, strong) UIImage *icon;

@end
```

Thanks to the magic of auto-synthesis, there's nothing else you need to do with that class. It holds all the various pieces of information about the published audio units, and an `AudioComponentDescription` describes the audio unit. Recall from a few pages ago that you created one of these data structures when publishing the effects app's audio unit.

Now add another class to the project called **SelectEffectViewController** as a subclass of **UITableViewController**. Open **SelectEffectViewController.h** and modify it as shown below:

```
#import <UIKit/UIKit.h>
```

```
@class SelectEffectViewController;
@class Effect;

@protocol SelectEffectViewControllerDelegate <NSObject>
- (void)selectEffectViewController:
            (SelectEffectViewController*)viewController
                  didSelectEffect:(Effect*)effect;
- (void)selectEffectViewControllerWantsToClose:
            (SelectEffectViewController*)viewController;
@end

@interface SelectEffectViewController : UITableViewController

@property (nonatomic, weak)
    id <SelectEffectViewControllerDelegate> delegate;

@end
```

The above code sets up a standard `UITableViewController` with a delegate that is told when it should close and when an effect has been selected.

Open **SelectEffectViewController.m** and add an instance variable to the implementation block like so:

```
@implementation SelectEffectViewController {
    NSArray *_effects;
}
```

This will hold all the effects that have been found on the system; the method you're about to add will discover these effects for you.

Next, add the following imports to the top of the file:

```
#import "Effect.h"

@import AudioUnit;
@import AudioToolbox;
```

Now add the following method:

```
- (void)refreshList {
    // 1
    NSMutableArray *effects = [NSMutableArray new];

    // 2
    AudioComponentDescription searchDesc =
        { kAudioUnitType_RemoteEffect, 0, 0, 0, 0 };
```

```
    // 3
    AudioComponent component = NULL;
    while ((component =
            AudioComponentFindNext(component, &searchDesc)))
    {
        // 4
        AudioComponentDescription description;
        OSStatus err = AudioComponentGetDescription(component,
                                                    &description);
        if (err) continue;

        // 5
        Effect *effect = [[Effect alloc] init];
        effect.componentDescription = description;
        effect.icon = AudioComponentGetIcon(component, 44.0f);

        CFStringRef name;
        AudioComponentCopyName(component, &name);
        effect.name = (__bridge NSString *)name;

        [effects addObject:effect];
    }

    // 6
    _effects = effects;
    [self.tableView reloadData];
}
```

This method generates the list of all the effects audio units that are available on the system. Here's what you do in each step:

1. Initialize a new array to hold the effects.
2. The `AudioComponentFindNext` (see next step) allows you to search through all available audio units. Here it takes a search description data structure set up with zero for all fields except for the component type — the first parameter. You only want to find the effects, so you pass `kAudioUnitType_RemoteEffect` as the first argument so that only those type of audio units will be returned.
3. Loop through all the available audio units using the search descriptor that you created in the previous step. The `component` variable is populated with the next component each time through the loop.
4. The description needs to be passed back through the `Effect` object; therefore make an attempt to get it, but if you encounter an error then scrap this audio unit continue with the loop.
5. Create the effect model object and set it up with relevant information.

6. Once all the audio units have been processed, update the instance variable with the new array and reload the table view.

Next, find `viewDidLoad` and add the following two lines to the end:

```
[self.tableView registerClass:[UITableViewCell class]
       forCellReuseIdentifier:@"Cell"];
[self refreshList];
```

Here you register a cell class and then call `refreshList` to ready the list of effects for display.

Finally, delete the starter method implementations for `numberOfSectionsInTableView:`, `tableView:cellForRowAtIndexPath:`, and `tableView:numberOfRowsInSection:`. In their stead, add all of the following methods:

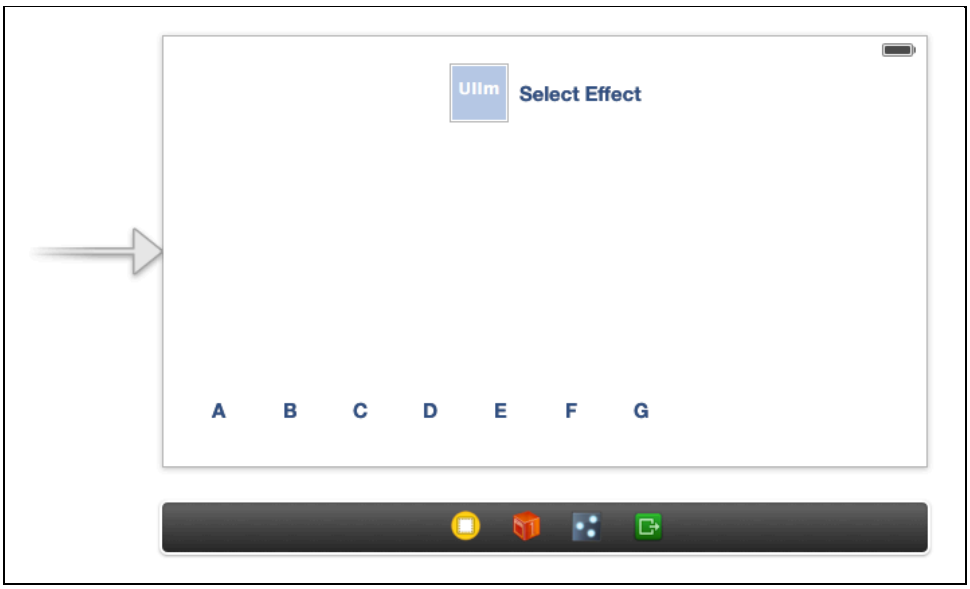
```
- (IBAction)closeTapped:(id)sender {
    [_delegate selectEffectViewControllerWantsToClose:self];
}

- (NSInteger)numberOfSectionsInTableView:
            (UITableView *)tableView
{
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    return _effects.count;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
      cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:@"Cell"
                             forIndexPath:indexPath];

    Effect *effect = _effects[indexPath.row];
    cell.imageView.image = effect.icon;
    cell.textLabel.text = effect.name;

    return cell;
}
```

```
- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    Effect *effect = _effects[indexPath.row];
    [_delegate selectEffectViewController:self
                            didSelectEffect:effect];
}
```

The first method above allows the view controller to be dismissed without selecting any effect in case the user wants to cancel the action. The other methods are all standard table view delegate and data source method implementations.

Now to make use of that table view controller. Open **Main.storyboard** and place a button in the top center of onto the lone view controller's view and title it "**Select Effect**". Next add a 44x44 image view to the left of the button, and then add an outlet named **effectIconImageView** to the view controller for the image view and wire it up.

Your scene should now look like this:

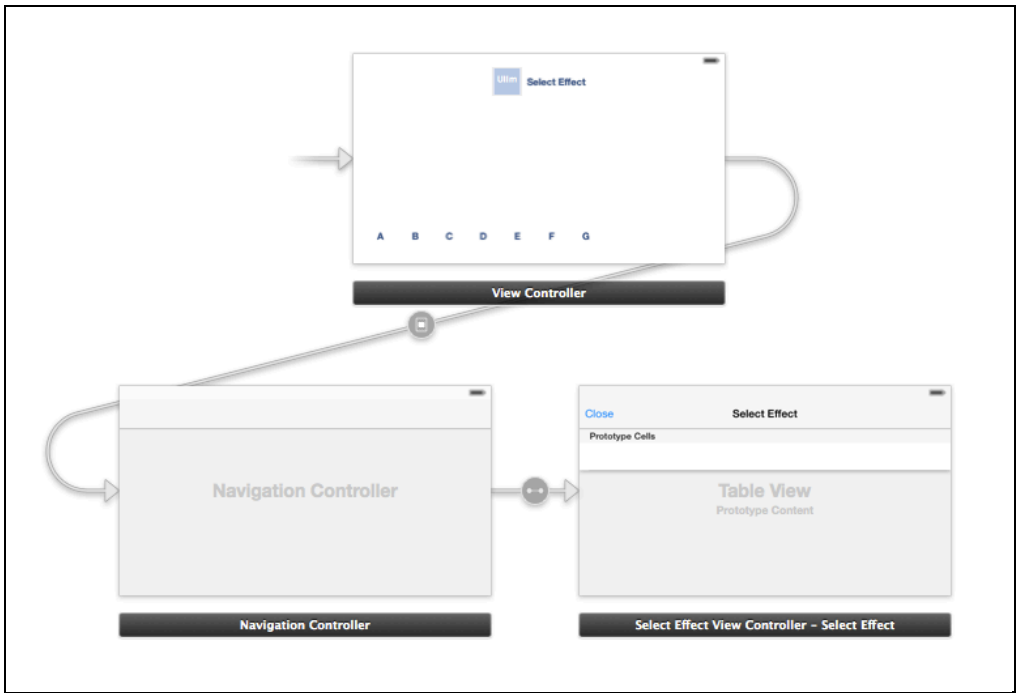


Next, drag a **Table View Controller** onto your scene. Make sure the table view controller is selected — not the table view — and change its orientation to **Landscape** in the **Attributes Inspector**. Next, change the class to **SelectEffectViewController** in the **Identity Inspector**.

Now select the new view controller you just added, click **Editor\Embed In\Navigation Controller** and change the navigation controller's orientation to landscape. Change the title of the select effect view controller to "**Select Effect**" and add a bar button item to the left position on the navigation bar. Change the button's title to **Close** and wire it up to the **closeTapped:** method on the view controller.

Finally, wire up the **Select Effect** button on the main view controller to perform a modal segue to the navigation controller. Give the segue the identifier **EffectSegue**.

That finishes off the modifications to the storyboard. Your scene should now look like this:



Now open **ViewController.m** and add the following imports:

```
#import "SelectEffectViewController.h"
#import "Effect.h"
```

Then make `ViewController` conform to the `SelectEffectViewControllerDelegate` protocol by modifying it as below:

```
@interface ViewController ()
    <GuitarNeckDelegate, SelectEffectViewControllerDelegate>
```

Then add the following method:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue
                 sender:(id)sender
{
    if ([segue.identifier isEqualToString:@"EffectSegue"]) {
        UINavigationController *navigationController =
    ((UINavigationController *)segue.destinationViewController);

        SelectEffectViewController *selectEffectViewController =
```

```
            (SelectEffectViewController*)
                navigationController.topViewController;

        selectEffectViewController.delegate = self;
    }
}
```

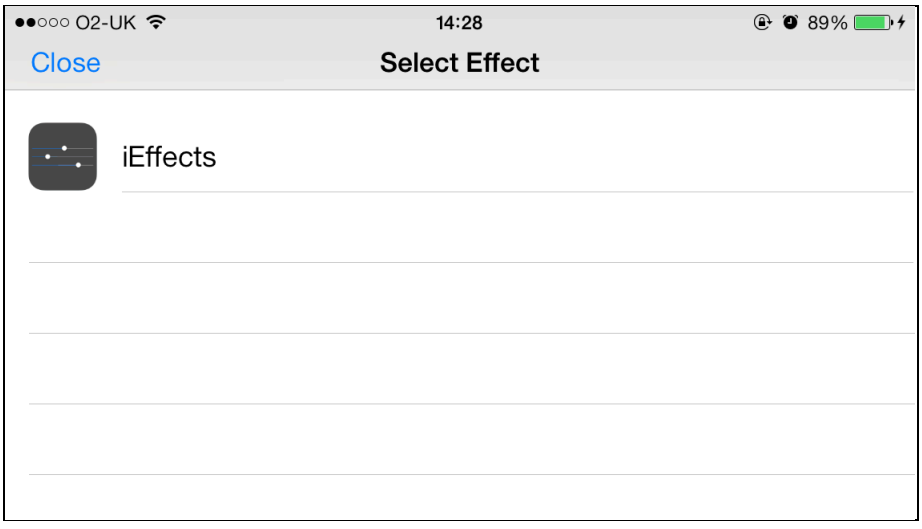
This method sets up the modal segue you added in the storyboard and sets the delegate of `SelectEffectViewController` so that it can inform [who?]when an effect is chosen.

Next, add the following two methods:

```
- (void)selectEffectViewControllerWantsToClose:
        (SelectEffectViewController*)viewController
{
    [self dismissViewControllerAnimated:YES completion:nil];
}

- (void)selectEffectViewController:
        (SelectEffectViewController*)viewController
                didSelectEffect:(Effect*)effect
{
    [self dismissViewControllerAnimated:YES completion:nil];
}
```

This implements the two required delegate methods of the effect picker. Right now, they don't do anything except dismiss the picker.

Build and run your app; if everything has gone according to plan then you should see iEffects in the list of available effects:

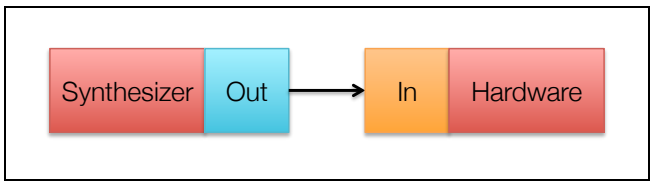


Wiring up the guitar effect

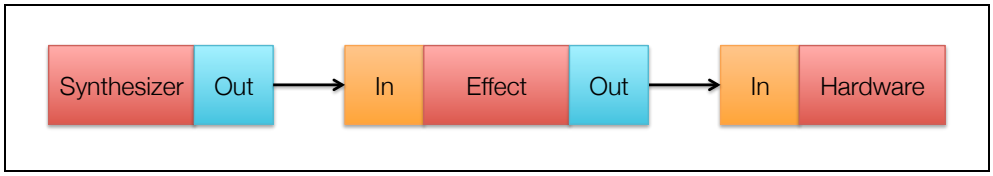
Now that users have a way to select the effect they want to use, the last thing to do is wire up the effect in the audio graph. In the following chapter you will learn about audio graphs; but for now all you need to know is that audio flows through an audio graph.

Each node in the graph can accept input, process the input, produce output, or even perform a combination of all three. As an example, an audio source such as a microphone will only produce output, while an audio sink, such as a speaker, will only accept input.

The image below represents the audio graph of the guitar app without any effects applied:



The synthesizer node (the element that creates the guitar sounds) has only one output, which connects to the hardware node's input, i.e., the speaker. When the effect is added to the graph, it will end up looking something like the diagram below:



The following steps are required to insert the effect into the audio processing chain:

1. Stop the graph from processing audio. This needs to be done whenever the graph is altered.
2. Disconnect the synthesizer from the hardware.
3. Connect the effect node's output to the hardware node's input.
4. Connect the synthesizer node's output to the effect node's input.
5. Start the graph.

> **Note:** It's customary — but not required — to wire up a graph in reverse, or right to left from the perspective of the diagram above. When dealing with complicated graphs that include mixers (multiple inputs, single output) the code is often cleaner and easier to read if you wire in reverse.

Open **ViewController.m** and add the following instance variables to the implementation block:

```
AudioUnit _effectUnit;
AUNode _effectNode;
```

These hold the references to the effect audio unit and associated node within the audio graph when an effect is connected. Next, add the following method:

```
- (void)connectEffect:(Effect*)effect {
    // 1
    [self stopAUGraph];

    // 2
    AUNode newEffectNode;
    AudioComponentDescription desc =
        effect.componentDescription;
    AUGraphAddNode(_audioGraph, &desc, &newEffectNode);

    if (newEffectNode) {
        // 3
        if (_effectNode) {
            AUGraphDisconnectNodeInput(_audioGraph,
                                       _effectNode,
                                       0);
            AUGraphRemoveNode(_audioGraph, _effectNode);
            _effectIconImageView.image = nil;
            _effectUnit = NULL;
        }

        _effectNode = newEffectNode;

        // 4
        AUGraphNodeInfo(_audioGraph,
                        _effectNode,
                        0,
                        &_effectUnit);

        // 5
        AudioUnitAddPropertyListener(_effectUnit,
                       kAudioUnitProperty_IsInterAppConnected,
                                     AudioUnitPropertyChanged,
                                     (__bridge void*)self);

        // 6
        AUGraphDisconnectNodeInput(_audioGraph, _ioNode, 0);
```

```
        // 7
        AUGraphConnectNodeInput(_audioGraph,
                                _effectNode,
                                0,
                                _ioNode,
                                0);

        // 8
        AUGraphConnectNodeInput(_audioGraph,
                                _synthNode,
                                0,
                                _effectNode,
                                0);

        // 9
        _connected = YES;
        _effectIconImageView.image = effect.icon;
    } else {
        NSLog(@"Failed to obtain effect audio unit.");
    }

    // 10
    [self startAUGraph];
    CAShow(_audioGraph);
}
```

This method wires in the chosen effect. Here's what it does step by step:

1. Stops the graph before performing any manipulation on it.
2. Gets the `AudioComponentDescription` from the effect, and turns it into a node in the graph using `AUGraphAddNode`.
3. Removes the old effect (if it exists) from the graph before `_effectNode` is switched to the new node.
4. Gets the `AudioUnit` from the `AUNode`. This is necessary for the next step.
5. Just as in the effects app, the property listener listens for changes to the inter-app audio state.
6. Disconnects the hardware node.
7. Connects the effect node's output to the hardware node's input.
8. Connects the synthesizer node's output to the effect node's input.
9. Updates the connection state and the icon image view.
10. Finally, restarts the graph. `CAShow` dumps the current state of the graph to the console to aid debugging.

Step 5 adds the property listener for state changes to inter-app audio. This is required such that if the connected effects app disconnects, the graph can be rewired back to its original state. The effects app might disconnect if it crashes or if the system kills it to free up some memory.

To implement the audio listener, find the class continuation category for **ViewController** and add the following method prototype:

```
- (void)audioUnitPropertyChanged:(void *)object
                            unit:(AudioUnit)unit
                          propID:(AudioUnitPropertyID)propID
                           scope:(AudioUnitScope)scope
                         element:(AudioUnitElement)element;
```

Now add the following function after the class continuation category, but before the implementation block:

```
void AudioUnitPropertyChanged(void *inRefCon,
                              AudioUnit inUnit,
                              AudioUnitPropertyID inID,
                              AudioUnitScope inScope,
                              AudioUnitElement inElement)
{
 ViewController *SELF = (__bridge ViewController *)inRefCon;
    [SELF audioUnitPropertyChanged:inRefCon
                              unit:inUnit
                            propID:inID
                             scope:inScope
                           element:inElement];
}
```

Just like in the effects app, the above C callback uses the opaque pointer `inRefCon` to perform the C to Objective-C dance.

Now, add the following method to the `ViewController` class:

```
- (void)audioUnitPropertyChanged:(void *)object
                            unit:(AudioUnit)unit
                          propID:(AudioUnitPropertyID)propID
                           scope:(AudioUnitScope)scope
                         element:(AudioUnitElement)element
{
    if (propID == kAudioUnitProperty_IsInterAppConnected) {
        UInt32 connected;
        UInt32 dataSize = sizeof(UInt32);
        AudioUnitGetProperty(_effectUnit,
                             kAudioUnitProperty_IsInterAppConnected,
```

```
                                kAudioUnitScope_Global,
                                0,
                                &connected,
                                &dataSize);

        _connected = (BOOL)connected;

        if (!_connected && _effectNode) {
            [self stopAUGraph];

            AUGraphDisconnectNodeInput(_audioGraph,
                                       _effectNode,
                                       0);
            AUGraphRemoveNode(_audioGraph, _effectNode);
            _effectIconImageView.image = nil;
            _effectNode = 0;
            _effectUnit = NULL;

            _effectIconImageView.image = nil;

            AUGraphConnectNodeInput(_audioGraph,
                                    _synthNode,
                                    0,
                                    _ioNode,
                                    0);
        }

        [self startStopGraphAsRequired];
    }
}
```

This code checks the connection state of the effects audio unit. If the effects unit isn't connected, it removes the effects node from the graph and rewires the synthesizer node straight to the hardware node.

You're almost there – there's just a couple of things left. First, add the following line to the end of `selectEffectViewController:didSelectEffect:`

```
[self connectEffect:effect];
```

This will ensure that the selected effect app is wired in.

Finally, add the following method:

```
- (IBAction)openEffectApp:(id)sender {
    if (_effectUnit) {
```

```
        CFURLRef url;
        UInt32 size = sizeof(url);
        OSStatus result =
            AudioUnitGetProperty(_effectUnit,
                                 kAudioUnitProperty_PeerURL,
                                 kAudioUnitScope_Global,
                                 0,
                                 &url,
                                 &size);
        if (result == noErr) {
            [[UIApplication sharedApplication]
                openURL:(__bridge NSURL*)url];
        }
    }
}
```

Just as in iEffects, you can open up the linked audio app directly; the method above will get the effect app's URL if it's connected.

Follow the same steps as before on the storyboard: set the image view's user interaction to **Enabled** and add a tap gesture recognizer. The gesture recognizer's action should then be wired up to this method.

That's it — build and run your app and try out your fully functional suite of inter-audio apps! Strum the guitar, then connect the iEffects app; you should hear the reverb effect added to the guitar sound as it's processed through the iEffects app.

> **Note:** You may find that the app crashes if you connect the effect while the audio is still playing. If you encounter this, just ensure you wait until the audio has finished before connecting the effects app. This seems to happen due to a bug in the synthesizer node.

The next chapter goes into more depth about Core Audio, audio graphs and interfacing with the hardware. So if you're looking to learn more details about Core Audio then carry on reading!

Challenges

Try out this challenge to get some practice with the techniques you've just learned.

Challenge 1: Change the effect

The iEffects app only has a single reverb effect, but there are many more effects available. In this challenge, edit the iEffects app to make use of a different effect. Here is a list of some of the others that are available:

- Low-pass filter
- High-pass filter
- Band-pass filter
- Delay
- Distortion

Tip: The various effects units are defined in `AUComponent.h`.

Chapter 26: Intermediate Inter-App Audio

By Matt Galloway

In the previous chapter you learned how to add inter-app audio to an existing app. However, you may have been a bit confused by some of the code in the starter project if you weren't already familiar with Core Audio.

In this chapter you will learn more about Core Audio, including what audio graphs are and how to wire one up to process audio. You'll also learn how Core Audio interfaces with the microphone and the speaker on iOS.

By the end of this chapter you'll have written a Core Audio app from scratch that uses inter-app audio as well. It will act as a host in the inter-app audio sense, by connecting an instrument to an effect.

Ready to dive in head first to Core Audio? Read on!

What is Core Audio?

Core Audio is a low-level C API for interfacing with audio hardware and processing audio in iOS. It is written in C, rather than Objective-C, as it needs to be an extremely low latency process. In fact, just the simple overhead of the Objective-C runtime could significantly impact the performance. It really is that low-level!

Not every developer needs to drop down to Core Audio. Apple has kindly wrapped up most of the common tasks into the AV Foundation framework. For example, `AVPlayer` provides a way to play a sound file. Doing the same thing directly in Core Audio would require a significant amount of code.

So when is direct Core Audio access a good idea? Well, if you want to create an instrument app, an app that processes audio, or an app that requires real-time audio such as VoIP, then you will likely need to drop down to Core Audio. For example, an app that takes input and mixes it with a sound file might use Core Audio since the two audio streams need to be in-sync, as even the tools in AV Foundation are often not good enough for this.

Audio units

One of the core concepts of Core Audio is the **audio unit**. These are objects that represent a single I/O or processing block. For example, to interface with the hardware you use what is known as the "remote IO audio unit" – more on that later. Another example is a mixer audio unit that can take multiple inputs, mix the audio and spit out a single output stream.

Apple provides several pre-defined audio units in the SDK, such as a voice processor that performs noise cancellation, but you can also create your own custom units. The SDK includes base classes for effect, instrument, and format conversion.

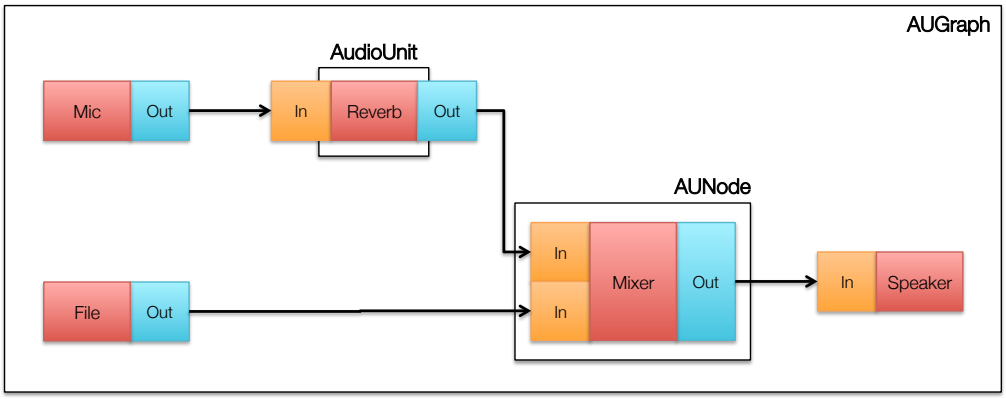
Audio units have one or more busses, which separate out logical streams within the audio unit. Most audio units will only have one bus as they accept one input and produce one output. However the hardware I/O unit (discussed later) has two busses – one for the microphone and one for the speaker.

Audio graphs

An **audio graph** is made up of many audio units. An audio unit is known as a **node** when it is part of an audio graph. You can think of audio flowing through an audio graph like water flowing through a pipe network.

The audio unit that generates audio in a graph is referred to as the **source**. In the water analogy, this would be the tap producing the raw water. Audio units that process audio can be compared to things that change the state of the water by heating it up, changing its color, or mixing it with water from other sources. The final audio unit in a graph is referred to as the **sink**. What this represents in the water analogy is self-explanatory!

An example audio graph is shown in the image below:



In this graph, there are five nodes. There are two source nodes, two processing nodes and one sink node. In the above audio graph, audio is taken from the microphone, reverb added, mixed with audio from a file and finally played out to the speaker.

In Core Audio terms, there are three objects that you need to be aware of when creating audio graphs:

- `AUGraph` – This is the object that contains the overall graph structure. You add nodes to it, wire the nodes up and then start audio processing. Changes can be made to the graph, but audio processing must be stopped before doing so, and then restarted once the changes have been made.
- `AUNode` – This is the node object you add to a graph and wire together with other nodes. A node is a wrapper for an audio unit and exposes inputs and outputs that are wired to form the graph.
- `AudioUnit` – This is the object where the actual audio processing happens. As explained earlier, audio units are either system-supplied or custom made.

Audio formats

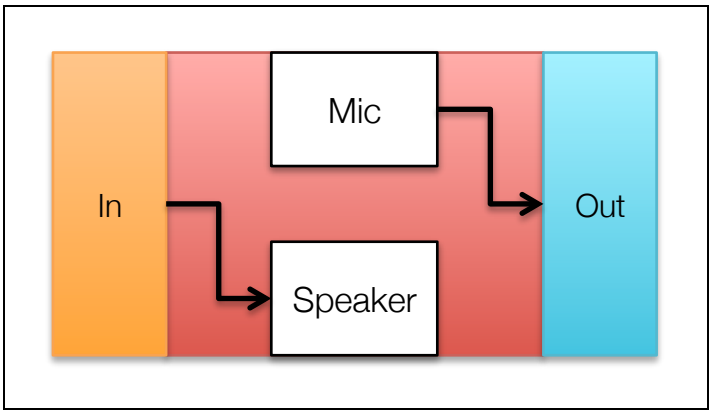
When dealing with a graph, you might need to consider audio formats. Audio stored in one format may be completely incompatible with audio stored in another format. So if one audio unit expects its input to be a certain format, then upstream units must supply that same format. Therefore, unless you have a unit that changes the audio format, all audio within a graph must be the same format.

Sometimes there are limitations on the audio format that your source can produce. For example, a microphone might only be able to provide a certain binary format at specific discrete sample frequencies. If there are discrepancies between two audio formats, then you can use a special conversion audio unit to convert one format to another. You might use this strategy when processing multiple audio streams, since you would require all audio streams to be of the same format.

Remote I/O

In iOS, you interface with the hardware (microphone and speaker) through the special remote I/O audio unit. This is different from other units, as the input and output are not connected like other audio units. Instead, the input is connected to the speaker and the microphone is connected to the output.

Yeah, that sounds a little odd, but the following diagram showing the internal workings of the remote I/O unit probably explains it a little better:



Sometimes it can get a bit confusing when using the remote I/O unit, because audio processing can both begin and end with the remote I/O unit. Therefore your graph looks like one big loop, even though it's a proper directed graph.

The magic behind the curtain is that the audio unit has two busses: one bus for the microphone (bus 1) and one for the speaker (bus 0). A good way to remember which bus is which is the following:

- The microphone is the Input, and **I** looks like a one
- The speaker is the Output, and **O** looks like a zero

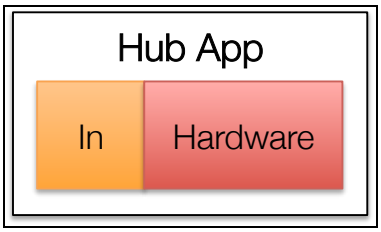
Creating a hub app

In this part of the tutorial you'll create an app similar to the one in the last chapter — but this time completely from scratch. It's a host app — or **hub** — that connects two other inter-app audio apps. This might seem a little redundant; why wouldn't you just connect the two apps directly?

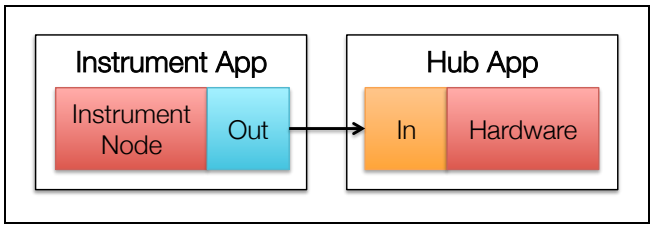
Connecting the two apps directly is fine for a simple application, but using a host app makes complicated unit wiring scenarios possible in apps that mix lots of different audio sources, such as GarageBand.

> **Note:** From this point onwards, the tutorial assumes you have read the previous chapter. If you have not yet read that then it's highly recommended that you go back and read it first.

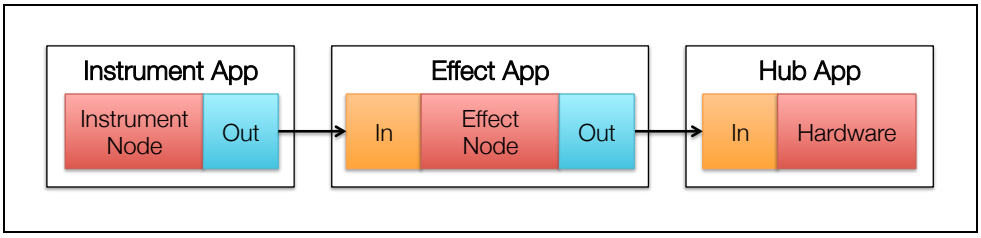
The graph for the app will have three states. The first state is when neither an instrument nor effect is connected. In this state there will be just one node — the hardware node. It won't be connected to anything and the graph won't be running. The graph will therefore look like this:



The second state is when there's only an instrument is connected, but no effect. In this state the graph would look like this:



The third state is when an instrument and an effect are both connected to the hub app. In this final state the graph will look like the following:



The instrument node will run inside the instrument app; likewise, the effect node will run inside the effect app. The only node running inside the hub app itself is the hardware node required to play audio out through the speaker.

> **Note:** You'll need the iGuitar and iEffects apps installed on your device so that this app can connect to them. If you didn't create those apps in the previous chapter, you'll find those two projects in the resources folder for this project.

Getting started

Open Xcode and create a new project called **AudioHost** from the **Single View Controller** template. Make it iPhone-only and save it wherever you wish.

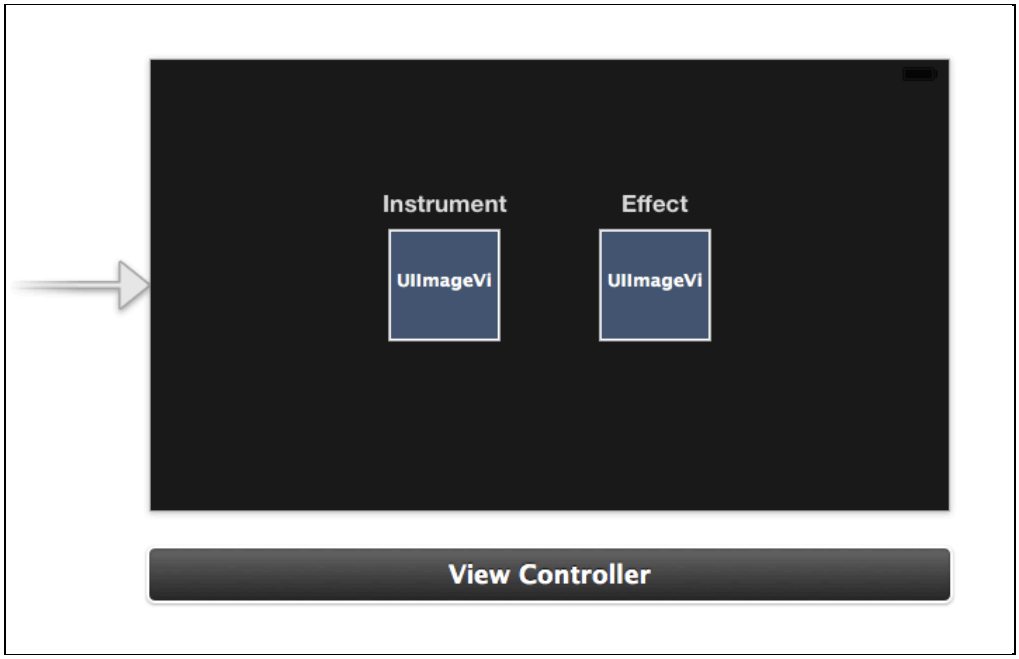
Before doing anything else, you'll need to set up inter-app audio. Open the project's **Capabilities** tab on the project info screen. Turn on **Inter-App Audio** and **Background Modes**, then tick **Audio & AirPlay**. Once again, hooray for Apple's new Capabilities tab! :]

Next, navigate to the **General** tab of the project settings and un-tick **Portrait** in the **Device Orientation** block. Just like iGuitar and iEffects, this app will only support landscape mode.

You'll find an icon in the resources for this chapter, called **AudioHost-Icon@2x.png** that you can use for the icon if you wish. Add it in the usual way as part of the assets catalog for the app.

Open **Main.storyboard** and add two labels and two image views to the view controller. Set the background color of the view to RGB (25, 25, 25) and set the labels' colors to RGB (224, 224, 224). Turn on **User Interaction Enabled** for both image views and set the views' background colors to RGB (224, 224, 224).

Modify the layout of your labels and image views to resemble the screenshot below:



Don't worry too much about using auto-layout to keep the views looking perfect when the main view resizes; just lay them out as you wish. This isn't a tutorial on auto-layout, after all!

Finally, wire up the two image views to outlets in `ViewController` called `instrumentIconImageView` and `effectIconImageView`.

That does it for setting up the UI; on to the audio graph implementation!

Creating the graph

Now for the fun Core Audio stuff! Open **ViewController.m** and add the following imports to the top of the file:

```
@import AudioToolbox;
```

```
@import AudioUnit;
@import AVFoundation;
```

Next, add the following instance variables to the implementation block:

```
@implementation ViewController {
    AUGraph _audioGraph;
    AudioUnit _ioUnit;
    AudioUnit _instrumentUnit;
    AudioUnit _effectUnit;
    AUNode _ioNode;
    AUNode _instrumentNode;
    AUNode _effectNode;
    BOOL _graphStarted;
    BOOL _connectedInstrument;
    BOOL _connectedEffect;
}
```

These hold the relevant audio graph, audio units and audio nodes for your app. As well, there are a few flags to determine whether the graph is started and whether instrument and effect nodes are connected.

Time to create the audio graph! Add the following method:

```
- (void)createAUGraph {
    // 1
    NewAUGraph(&_audioGraph);

    // 2
    AudioComponentDescription iOUnitDescription;
    iOUnitDescription.componentType =
        kAudioUnitType_Output;
    iOUnitDescription.componentSubType =
        kAudioUnitSubType_RemoteIO;
    iOUnitDescription.componentManufacturer =
        kAudioUnitManufacturer_Apple;
    iOUnitDescription.componentFlags = 0;
    iOUnitDescription.componentFlagsMask = 0;
    AUGraphAddNode(_audioGraph, &iOUnitDescription, &_ioNode);

    // 3
    AUGraphOpen(_audioGraph);

    // 4
    AUGraphNodeInfo(_audioGraph, _ioNode, NULL, &_ioUnit);
```

```
    // 5
    AudioStreamBasicDescription format;
    format.mChannelsPerFrame = 2;
    format.mSampleRate =
        [[AVAudioSession sharedInstance] sampleRate];
    format.mFormatID = kAudioFormatLinearPCM;
    format.mFormatFlags =
        kAudioFormatFlagsNativeFloatPacked |
        kAudioFormatFlagIsNonInterleaved;
    format.mBytesPerFrame = sizeof(Float32);
    format.mBytesPerPacket = sizeof(Float32);
    format.mBitsPerChannel = 32;
    format.mFramesPerPacket = 1;

    AudioUnitSetProperty(_ioUnit,
                         kAudioUnitProperty_StreamFormat,
                         kAudioUnitScope_Output,
                         1,
                         &format,
                         sizeof(format));

    AudioUnitSetProperty(_ioUnit,
                         kAudioUnitProperty_StreamFormat,
                         kAudioUnitScope_Input,
                         0,
                         &format,
                         sizeof(format));

    CAShow(_audioGraph);
}
```

Here's what you do in the code above, comment by comment:

1. Initialize the graph object. Almost all Core Audio APIs return a status code to indicate success or failure, so passing a variable by reference that gets filled in (an "out parameter") is used instead.
2. `AudioComponentDescription` structures describe audio units. Here you ask a graph to add a node with one of these structures. The only node in the graph at this point is the remote IO node. The instrument and effect nodes will be added when the user selects them.
3. Now that all the nodes have been added, the graph can be opened with `AUGraphOpen`. At this point, the nodes and their associated audio units will be ready and waiting to be set up as required.
4. Get the remote IO audio unit from the node so that you can apply formats to it in the next step. Once again the out parameters provide the required units to you.

5. Set both the input and output formats. You don't have to set the output format because you're not using the node's output in this app, but sometimes audio will stop working if both formats aren't configured. To be safe, set both formats and the app will be fine even if you never use the output.

Next, change `viewDidLoad` as follows:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    [self createAUGraph];
}
```

This ensures that the graph is created when the view is loaded.

Build and run your app, and your UI will appear. It doesn't do a whole lot right now, but this is a good checkpoint to ensure your code compiles and runs correctly.

Take a look at the console, and you'll see the following output:

```
AudioUnitGraph 0x2C3000:
  Member Nodes:
    node 1: 'auou' 'rioc' 'appl', instance 0xa144140 O
  CurrentState:
    mLastUpdateError=0, eventsToProcess=F, isRunning=F
```

This tells you that there is only one node (`node 1`) and the graph is not yet running (`isRunning` is false). Everything checks out so far.

When you dynamically rewire audio graphs, you need to start and stop the graph accordingly; the next section shows you how to do this.

Starting and stopping the graph

Open **ViewController.m** and add the following methods:

```
- (void)startAudioSession {
    AVAudioSession *session = [AVAudioSession sharedInstance];
    [session setPreferredSampleRate:
        [session sampleRate] error:nil];
    [session setCategory:AVAudioSessionCategoryPlayback
          withOptions:AVAudioSessionCategoryOptionMixWithOthers
                error:nil];
    [session setActive:YES error:nil];
}

- (void)startStopGraphAsRequired {
    if (_connectedInstrument) {
        [self startAUGraph];
```

```
    } else {
        [self stopAUGraph];
    }
}

- (void)startAUGraph {
    if (!_graphStarted && _audioGraph) {
        [self startAudioSession];

        Boolean outIsInitialized;
        AUGraphIsInitialized(_audioGraph, &outIsInitialized);
        if (!outIsInitialized) {
            AUGraphInitialize(_audioGraph);
        }

        AUGraphStart(_audioGraph);

        _graphStarted = YES;
    }
}

- (void)stopAUGraph {
    if (_graphStarted && _audioGraph) {
        AUGraphStop(_audioGraph);

        Boolean outIsInitialized;
        AUGraphIsInitialized(_audioGraph, &outIsInitialized);
        if (outIsInitialized) {
            AUGraphUninitialize(_audioGraph);
        }

        _graphStarted = NO;
    }
}
```

If you followed along in the previous chapter, these methods should look familiar. They are very similar to the equivalent methods in iGuitar and iEffects that handle starting and stopping the graph as required, as well as starting an audio session with iOS.

Selecting the instrument and effect

Next you need a way to select the instrument and effect. iEffects uses a modal view controller to search and list all available effects and prompt the user to select the desired effect. You can easily extend that functionality to allow selection of both instruments and effects.

Add a class to the project called `InterAppAudioUnit`, making it a subclass of **NSObject**. Then open **InterAppAudioUnit.h** and modify it so it looks like the code below:

```
#import <Foundation/Foundation.h>

@import AudioUnit.AudioComponent;

@interface InterAppAudioUnit : NSObject

@property (nonatomic, strong) NSString *name;
@property (nonatomic, assign)
    AudioComponentDescription componentDescription;
@property (nonatomic, strong) UIImage *icon;

@end
```

These properties will contain the details about the selected inter-app audio unit. If you completed the previous chapter, you'll notice that this is just the `Effect` class from iGuitar. Plus, thanks to auto-synthesis, nothing more needs to be done to the implementation of `InterAppAudioUnit`.

Next up is the UI for selecting an instrument or effect. Add a new class called `SelectIAAUViewController` to the project, making it a subclass of `UITableViewController`. Then open **SelectIAAUViewController.h** and modify it to look like this:

```
#import <UIKit/UIKit.h>

@import AudioUnit.AudioComponent;

@class SelectIAAUViewController;
@class InterAppAudioUnit;

@protocol SelectIAAUViewControllerDelegate <NSObject>
- (void)selectIAAUViewController:
        (SelectIAAUViewController*)viewController
                  didSelectUnit:(InterAppAudioUnit*)unit;
- (void)selectIAAUViewControllerWantsToClose:
        (SelectIAAUViewController*)viewController;
@end

@interface SelectIAAUViewController : UITableViewController

- (instancetype)initWithSearchDescription:
                (AudioComponentDescription)description;
```

```
@property (nonatomic, weak)
    id <SelectIAAUViewControllerDelegate> delegate;

@end
```

This is very similar to the `SelectEffectViewController` class of iGuitar. The only difference is that this contains an initializer method to pass in a search description. This allows the user to select either an instrument or effect from the view controller. The search description defines the type of audio units that are listed in the table.

Time to implement the rest of this class. Open **SelectIAAUViewController.m** and add the following imports to the top of the file:

```
#import "InterAppAudioUnit.h"

@import AudioUnit;
@import AudioToolbox;
```

Now add the following instance variables to the implementation block:

```
@implementation SelectIAAUViewController {
    AudioComponentDescription _searchDesc;
    NSArray *_units;
}
```

`_searchDesc` is populated during initialization (recall the initializer defined in the header) and `_units` holds the list of audio units from which the user can choose.

Next, add the following method:

```
- (void)refreshList {
    NSMutableArray *units = [NSMutableArray new];

    // 1
    AudioComponentDescription searchDesc = _searchDesc;
    AudioComponent component = NULL;
    while ((component =
            AudioComponentFindNext(component, &searchDesc)))
    {
        // 2
        AudioComponentDescription description;
        OSStatus err = AudioComponentGetDescription(component,
                                                    &description);
        if (err) continue;
```

```
        // 3
        InterAppAudioUnit *unit =
            [[InterAppAudioUnit alloc] init];
        unit.componentDescription = description;
        unit.icon = AudioComponentGetIcon(component, 44.0f);

        CFStringRef name;
        AudioComponentCopyName(component, &name);
        unit.name = (__bridge NSString *)name;

        [units addObject:unit];
    }

    // 4
    _units = units;
    [self.tableView reloadData];
}
```

This method searches for all available audio units and displays them to the user. It's very similar to the equivalent method in iGuitar's `SelectEffectViewController`, except here the user of the class sets the search description. Comment by comment, here's a rundown of the method's actions:

1. Make a copy of the search description since some of the private flags will be changed in the loop. Call `AudioComponentFindNext` to iterate through the discovered audio units matching the search description.
2. Get the description of the audio unit, and in the rare case that this call fails, continue the loop without further processing of that audio unit. It's always worth handling the error case.
3. Initialize a new `InterAppAudioUnit` object for each discovered audio unit with its description, icon and name. Add the audio unit to the `units` array.
4. Finally, once all audio units have been collected, set the instance variable and reload the table view.

Now, find `viewDidLoad` and modify it as follows:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    self.navigationItem.leftBarButtonItem =
        [[UIBarButtonItem alloc] initWithTitle:@"Cancel"
                                         style:UIBarButtonItemStyleBordered
                                        target:self
                                        action:@selector(closeTapped:)];
    [self.tableView registerClass:[UITableViewCell class]
           forCellReuseIdentifier:@"Cell"];
    [self refreshList];
```

```
}
```

This adds a button to the navigation bar to give the user an option to cancel the selection process. It also registers a standard cell for the table view and kicks off a refresh of the list of audio units.

Next, add the following method:

```
- (void)closeTapped:(id)sender {
    [_delegate selectIAAUViewControllerWantsToClose:self];
}
```

Tapping the cancel button calls the above method, which simply calls the delegate method that informs the delegate that the view controller will close without selecting an audio unit.

You now need to implement the custom initializer you declared earlier in the header. Add the following method:

```
- (instancetype)initWithSearchDescription:
                (AudioComponentDescription)description
{
    if ((self = [super initWithStyle:UITableViewStylePlain])) {
        _searchDesc = description;
    }
    return self;
}
```

This calls through to the superclass's (`UITableViewController`) designated initializer `initWithStyle:`. It also sets the instance variable that holds the search description used by `refreshList`.

When you create your own designated initializer as you did above, you need to ensure that you override the super-class's designated initializer. This ensures that when someone uses the super-class's designated initializer, your class is still initialized correctly.

Open **SelectIAAUViewController.m** and modify `initWithStyle:` as follows:

```
- (instancetype)initWithStyle:(UITableViewStyle)style {
    return [self initWithSearchDescription:
                (AudioComponentDescription){0}];
}
```

In this case, the view controller just needs to pass an empty search description through to the custom designated initializer. Sure, the `style` parameter will be ignored, but that's fine because you always want the style to be plain.

Finally, replace all of the table view delegate and data source methods at the bottom of the file with the following:

```
- (NSInteger)numberOfSectionsInTableView:
            (UITableView *)tableView
{
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    return _units.count;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:@"Cell"
                                        forIndexPath:indexPath];

    InterAppAudioUnit *unit = _units[indexPath.row];
    cell.imageView.image = unit.icon;
    cell.textLabel.text = unit.name;

    return cell;
}

- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    InterAppAudioUnit *unit = _units[indexPath.row];
    [_delegate selectIAAUViewController:self
                          didSelectUnit:unit];
}
```

These methods set up the table view to display the discovered audio units. When a row is selected, they call the delegate and inform it which row has been selected.

Now that you have the view controller for selecting both the instrument and the effect, it's time to wire these into the main view controller. Open **ViewController.m** and add the following two imports to the top of the file:

```
#import "SelectIAAUViewController.h"
#import "InterAppAudioUnit.h"
```

Then declare that `ViewController` implements the `SelectIAAUViewControllerDelegate` by editing the class-continuation category as follows:

```
@interface ViewController () <SelectIAAUViewControllerDelegate>
```

Next, add two instance variables to the implementation block as follows:

```
SelectIAAUViewController *_instrumentSelectViewController;
SelectIAAUViewController *_effectSelectViewController;
```

These variables hold the two view controllers for selecting the instrument and effect. Storing the view controller references in variables helps you determine which view controller is calling back when the delegate methods are called.

Next, add the following two methods, which make up the `SelectIAAUViewControllerDelegate` protocol you added earlier:

```
- (void)selectIAAUViewControllerWantsToClose:
        (SelectIAAUViewController *)viewController
{
    [self dismissViewControllerAnimated:YES completion:nil];
}

- (void)selectIAAUViewController:
        (SelectIAAUViewController *)viewController
                    didSelectUnit:(InterAppAudioUnit *)unit
{
    [self dismissViewControllerAnimated:YES completion:nil];
}
```

For now, this doesn't actually perform any rewiring of the graph. You'll get on to that in a moment.

Finally, you need a couple of methods to allow selection of an instrument and effect. Add the following two methods:

```
- (IBAction)selectInstrument:(id)sender {
    AudioComponentDescription description =
        { kAudioUnitType_RemoteInstrument, 0, 0, 0, 0 };
    _instrumentSelectViewController =
        [[SelectIAAUViewController alloc]
            initWithSearchDescription:description];
    _instrumentSelectViewController.delegate = self;

    UINavigationController *navController =
        [[UINavigationController alloc]
```

```
    initWithRootViewController:_instrumentSelectViewController];
    [self presentViewController:navController
                       animated:YES
                     completion:nil];
}

- (IBAction)selectEffect:(id)sender {
    AudioComponentDescription description =
        { kAudioUnitType_RemoteEffect, 0, 0, 0, 0 };
    _effectSelectViewController =
        [[SelectIAAUViewController alloc]
            initWithSearchDescription:description];
    _effectSelectViewController.delegate = self;

    UINavigationController *navController =
        [[UINavigationController alloc]
        initWithRootViewController:_effectSelectViewController];
    [self presentViewController:navController
                       animated:YES
                     completion:nil];
}
```

These methods create the selection view controllers and present them modally in a navigation controller. Notice how `AudioComponentDescription` only sets the audio unit type in each case. For the instrument selection you want just remote instruments; for the effect you want just remote effects.

Switch over to the storyboard and add tap gesture recognizers to each of the image views. Wire up the instrument icon's gesture recognizer to call `selectInstrument:` and the effect icon's gesture recognizer to call `selectEffect:`.

Build and run your app; tap on either of the two image views to select either an instrument or an effect. The instrument list will be empty, but iEffects should show up in the list of effects.

Of course, nothing will happen when you select it just yet! You need to dynamically rewire the graph before anything will happen.

Re-wiring the graph

Add the following method (still in **ViewController.m**):

```
- (void)connectInstrument:(InterAppAudioUnit*)unit {
    // 1
    [self stopAUGraph];

    // 2
```

```
    AUNode newInstrumentNode;
    AudioComponentDescription desc = unit.componentDescription;
    AUGraphAddNode(_audioGraph, &desc, &newInstrumentNode);

    // 3
    if (newInstrumentNode) {
        // 4
        if (_instrumentNode) {
            AUGraphDisconnectNodeInput(_audioGraph,
                                       _instrumentNode,
                                       0);
            AUGraphRemoveNode(_audioGraph, _instrumentNode);
            _instrumentIconImageView.image = nil;
            _instrumentUnit = NULL;
        }

        // 5
        _instrumentNode = newInstrumentNode;

        // 6
        AUGraphNodeInfo(_audioGraph,
                        _instrumentNode,
                        0,
                        &_instrumentUnit);

        // 7
        if (_effectNode) {
            AUGraphConnectNodeInput(_audioGraph,
                                    _instrumentNode,
                                    0,
                                    _effectNode,
                                    0);
        } else {
            AUGraphConnectNodeInput(_audioGraph,
                                    _instrumentNode,
                                    0,
                                    _ioNode,
                                    0);
        }

        // 8
        _connectedInstrument = YES;
        _instrumentIconImageView.image = unit.icon;
    } else {
        NSLog(@"Failed to obtain instrument audio unit.");
```

```
    }

    // 9
    [self startStopGraphAsRequired];
    CAShow(_audioGraph);
}
```

This method connects an instrument to the audio graph. If there is no effect connected, the instrument will connect directly to the hardware node. If there is an effect connected, the instrument is connected to the effect, which is in turn connected to the hardware node.

Here's a step-by-step of how the connection method works:

1. Stop the graph before rewiring it.
2. Create a node for the new instrument and add it to the audio graph.
3. If the node was created successfully, then the connection can continue. Otherwise it's a failure.
4. If there's already an instrument connected, remove the node and clean up.
5. Update the instance variable to hold the new instrument node.
6. `AUGraphNodeInfo` obtains the audio unit for the newly added node, just as you've seen previously.
7. If there's an effect node, then the instrument node needs to be wired to that; otherwise the instrument node is wired to the hardware node.
8. Update the instance variables to reflect the new state.
9. Finally, start the graph and log the graph state to the console.

That's the instrument connection — now to add an effect! Add the following method:

```
- (void)connectEffect:(InterAppAudioUnit*)unit {
    if (!_connectedInstrument) {
        UIAlertView *alert = [[UIAlertView alloc]
        initWithTitle:@"Error!"
              message:@"You need to select an instrument first!"
             delegate:nil
    cancelButtonTitle:nil
    otherButtonTitles:@"OK", nil];
        [alert show];
    }

    [self stopAUGraph];

    AUNode newEffectNode;
```

```
    AudioComponentDescription desc = unit.componentDescription;
    AUGraphAddNode(_audioGraph, &desc, &newEffectNode);

    if (newEffectNode) {
        if (_effectNode) {
            AUGraphDisconnectNodeInput(_audioGraph,
                                       _effectNode,
                                       0);
            AUGraphRemoveNode(_audioGraph, _effectNode);
            _effectIconImageView.image = nil;
            _effectUnit = NULL;
        }

        _effectNode = newEffectNode;

        AUGraphNodeInfo(_audioGraph,
                        _effectNode,
                        0,
                        &_effectUnit);

        AUGraphDisconnectNodeInput(_audioGraph, _ioNode, 0);

        AUGraphConnectNodeInput(_audioGraph,
                                _effectNode,
                                0,
                                _ioNode,
                                0);

        AUGraphConnectNodeInput(_audioGraph,
                                _instrumentNode,
                                0,
                                _effectNode,
                                0);

        _connectedEffect = YES;
        _effectIconImageView.image = unit.icon;
    } else {
        NSLog(@"Failed to obtain effect audio unit.");
    }

    [self startStopGraphAsRequired];
    CAShow(_audioGraph);
}
```

This method is very similar to the instrument connection method. The difference is in the wiring that needs to be done. As an exercise, read through the code and try

to explain it yourself using the instrument connection method as a guide. Remember that the effect node sits between the instrument node and the hardware node.

In the words of Steve Jobs: "one more thing". Add the following code to the top of `selectIAAUViewController:didSelectUnit:`

```
if (viewController == _instrumentSelectViewController) {
    [self connectInstrument:unit];
} else if (viewController == _effectSelectViewController) {
    [self connectEffect:unit];
}
```

This detects which view controller is indicating that an audio unit has been selected and calls the relevant connection method.

Build and run on your device, select iEffect as your effect, and then select an instrument...uh oh. When you try to select an instrument you won't see iGuitar in the list!

The issue is that iGuitar hasn't been declared as providing an instrument audio unit.

In the previous chapter iEffects was written to publish its audio unit, but iGuitar didn't need to, as iGuitar served as the host app and didn't need to publish any audio units. But now it's acting as a node app and therefore it *does* need to publish an audio unit if it wants to show up in the list.

Publishing iGuitar's audio unit

You now need to alter iGuitar slightly so that it can act as an instrument for inter-app audio purposes. Start off by opening the iGuitar project; either use your final project from the previous chapter, or use the one provided in the resources for this chapter.

If you've read the previous chapter you'll know where to start with this. That's right: add a key to the **Info.plist** file and then publish the node when the app launches.

Find the **iGuitar-Info.plist** file in the project navigator under **Supporting Files**. Right click on it and select **Open As\Source Code**. Then add the following at the bottom of the file, just before the `</dict>`:

```
<key>AudioComponents</key>
<array>
    <dict>
        <key>manufacturer</key>
        <string>i7bt</string>
        <key>name</key>
        <string>iGuitar</string>
```

```
        <key>type</key>
        <string>auri</string>
        <key>subtype</key>
        <string>iasp</string>
        <key>version</key>
        <integer>1</integer>
    </dict>
</array>
```

This is similar to what you added to **Info.plist** for iEffects; the only difference is the `name` and `type` values. The name is, of course, `iGuitar`, while the type is `auri`, which signifies an instrument.

Next, open **ViewController.m** and add the following method:

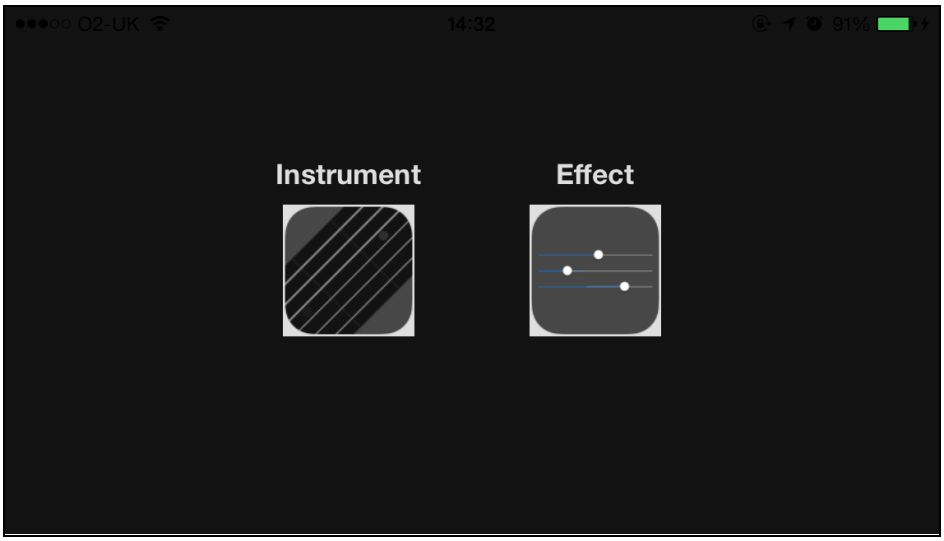
```
- (void)publishAsNode {
    AudioComponentDescription desc = {
        kAudioUnitType_RemoteInstrument,
        'iasp',
        'i7bt',
        0,
        0
    };
 AudioOutputUnitPublish(&desc, CFSTR("iGuitar"), 1, _ioUnit);
}
```

This publishes the app's remote IO audio unit to take part in inter-app audio. Note that the description and name are the same as were defined in the Info.plist file. This is required so that iOS knows that you're publishing the same node that was declared.

Finally, find `viewDidLoad` and add the following line directly after the call to `createAUGraph`:

```
[self publishAsNode];
```

That's it — build and run iGuitar, then switch to your AudioHost app and select an instrument. You should now see iGuitar in the list. Moreover, you should be able to select iGuitar along with an effect, then switch to iGuitar and start strumming.



The audio processing follows the sequence below:

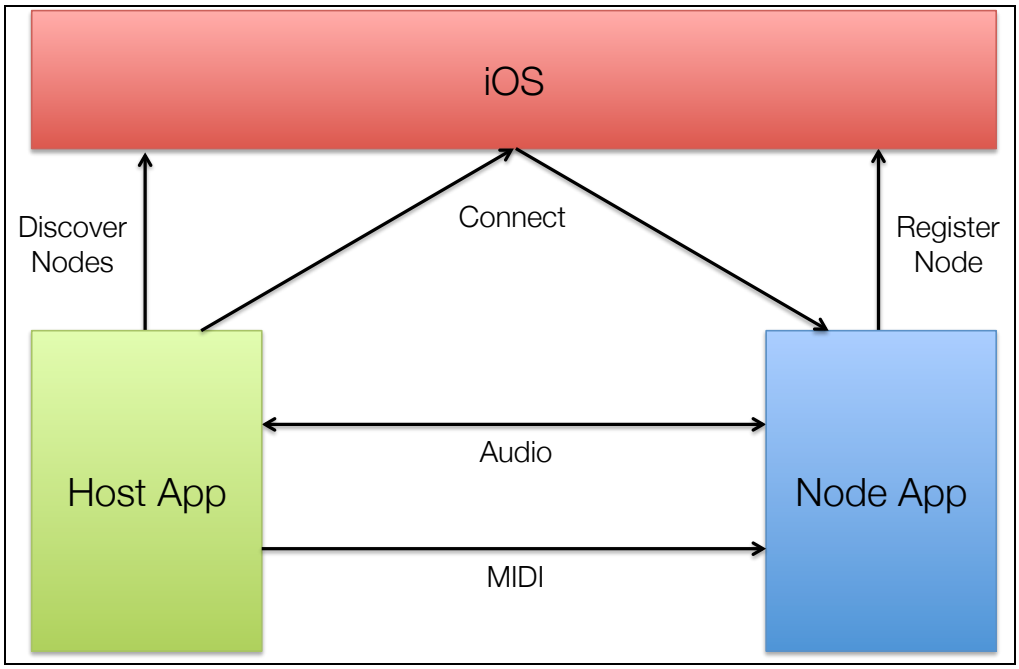
1. Created in iGuitar from the synthesizer.
2. Passed out from iGuitar to AudioHost.
3. Passed out from AudioHost to iEffects.
4. Processed by the effects unit in iEffects.
5. Passed out from iEffects to AudioHost.
6. Sent to the speaker hardware.

Complicated, but it works!

> **Note:** You may want to switch to iEffects and change a few sliders to modify the effect; this should help convince you that audio is flowing correctly through the three apps.

Sending MIDI events

The previous chapter noted that you could send more than just audio data; in particular you can send MIDI data, but note that only hosts can send MIDI events. Adding MIDI to your flow diagram results in the following:



Only instrument inter-app audio nodes can receive MIDI events from their host app. You're now going to add the ability for AudioHost to send MIDI events to iGuitar to make it play on its own as an example of how this callback mechanism works.

Setting up iGuitar to receive MIDI events

First you need to modify iGuitar to be able to receive MIDI events. Just as you set up property listeners to listen for changes to the inter-app audio connection state, you'll handle MIDI events in a similar manner. In the node app, you simply register a C function as a callback that's called every time a MIDI event is received.

Open the **iGuitar** project and open **ViewController.m**. Add the following code to `createAUGraph` just above the final `CAShow` call:

```
AudioUnitAddPropertyListener(_ioUnit,
                         kAudioUnitProperty_IsInterAppConnected,
                             AudioUnitPropertyChanged,
                             (__bridge void*)self);

AudioOutputUnitMIDICallbacks callbacks;
callbacks.userData = (__bridge void*)self;
callbacks.MIDIEventProc = AudioUnitMIDIEvent;
callbacks.MIDISysExProc = NULL;
AudioUnitSetProperty(_ioUnit,
                     kAudioOutputUnitProperty_MIDICallbacks,
                     kAudioUnitScope_Global,
                     0,
                     &callbacks,
```

```
                                sizeof(callbacks));
```

The `AudioUnitAddPropertyListener` call adds a property listener to the remote IO unit for changes to the inter-app connection state. This app's audio graph is currently only running when the app is in the foreground or an effect is connected to it. Now that it can be controlled from the AudioHost app, you need the graph to run in this new state.

The `AudioUnitSetProperty` call adds the MIDI event listener. It does this through the same Core Audio function that you saw earlier for setting audio formats. It declares that the function `AudioUnitMIDIEvent` will be called every time a MIDI event is received.

The event listener is registered on the global scope of the remote IO unit. Global scope is used when the property being set relates to neither input nor output. This is the case with MIDI events as they don't directly relate to either the microphone or the speaker — they're control messages.

Note that you're re-using an existing method for the property listener on the remote IO unit's inter-app connection state changes (`AudioUnitPropertyChanged`). If you need to remind yourself of how that works, check back over `AudioUnitPropertyChanged` and `audioUnitPropertyChanged:unit:propID:scope:element:`.

To handle the remote IO property listener, find `audioUnitPropertyChanged:unit:propID:scope:element:` and add another `if` statement around the existing `if` statement as shown below:

```
if (propID == kAudioUnitProperty_IsInterAppConnected) {
    /* Existing code used to be here… */
    if (unit == _effectUnit) {
        /* …but was moved here */
    }
}
```

Now, add an `else-if` block to that new `if`-statement as follows:

```
if (unit == _effectUnit) {
    /* Existing code */
} else if (unit == _ioUnit) {
    UInt32 connected;
    UInt32 dataSize = sizeof(UInt32);
    AudioUnitGetProperty(_ioUnit,
                         kAudioUnitProperty_IsInterAppConnected,
                         kAudioUnitScope_Global,
                         0,
                         &connected,
```

```
                            &dataSize);

    _connected = (BOOL)connected;

    [self startStopGraphAsRequired];
}
```

This handles the property listener requirements of the remote IO unit. Here, you're getting the current connection state and then calling `startStopGraphAsRequired` to ensure that the graph is started when an inter-app audio host app is connected.

Now it's time to add the MIDI event callback. Add the following method prototype to the class-continuation category:

```
- (void)audioUnitMIDIEvent:(void *)object
                    status:(UInt32)status
                     data1:(UInt32)data1
                     data2:(UInt32)data2
          offsetSampleFrame:(UInt32)offsetSampleFrame;
```

Next, add the following C function after the class-continuation category, before the implementation block (it should sit just below the `AudioUnitPropertyChanged` function):

```
void AudioUnitMIDIEvent(void *userData,
                        UInt32 inStatus,
                        UInt32 inData1,
                        UInt32 inData2,
                        UInt32 inOffsetSampleFrame)
{
    ViewController *SELF = (__bridge ViewController *)userData;
    [SELF audioUnitMIDIEvent:userData
                      status:inStatus
                       data1:inData1
                       data2:inData2
           offsetSampleFrame:inOffsetSampleFrame];
}
```

Just like the inter-app audio property listener, this uses the opaque pointer that can be set in the callback to pass the `ViewController` instance. This casts the opaque pointer to a `ViewController` instance and calls the Objective-C method on it. This is yet another example of the C to Objective-C dance that often happens with these kinds of callbacks.

Finally, add the following method:

```
- (void)audioUnitMIDIEvent:(void *)object
```

```
                      status:(UInt32)status
                       data1:(UInt32)data1
                       data2:(UInt32)data2
            offsetSampleFrame:(UInt32)offsetSampleFrame
{
    MusicDeviceMIDIEvent(_synthUnit,
                         status,
                         data1,
                         data2,
                         offsetSampleFrame);
}
```

It's deceivingly simple; all the code above does is pass the MIDI event straight through to the synthesizer unit.

Setting up AudioHost to send MIDI events

Now that iGuitar can receive MIDI events, you need to modify AudioHost to send some. You could add a full guitar neck interface to the AudioHost app, but that's a bit of overkill for this tutorial. So you're just going to add a button that plays a single note.

Open the **AudioHost** project and open **ViewController.m**. Then add the following method:

```
- (IBAction)playNote:(id)sender {
    if (_instrumentUnit) {
        UInt32 noteOnCommand = (0x9 << 4) | 0;
        MusicDeviceMIDIEvent(_instrumentUnit,
                             noteOnCommand,
                             60,
                             100,
                             0);

        double delayInSeconds = 2.0;
        dispatch_time_t popTime =
            dispatch_time(DISPATCH_TIME_NOW,
                          (int64_t)(delayInSeconds * NSEC_PER_SEC));

        dispatch_after(popTime,
                       dispatch_get_main_queue(),
                       ^(void){
            if (_instrumentUnit) {
                UInt32 noteOffCommand = (0x8 << 4) | 0;
                MusicDeviceMIDIEvent(_instrumentUnit,
                                     noteOffCommand,
```

```
                                        60,
                                        100,
                                        0);
            }
        });
    }
}
```

This is the method that sends the MIDI events. The note commands are just constants standing for “note on” and “note off”. You simply send MIDI events to the instrument audio unit, just as you would send MIDI events to a synthesizer audio unit.

The `dispatch_after` is used in the above code to send a MIDI event to stop playing the note after 2 seconds. This needs to be done to ensure that the note doesn't continue playing forever.

Now you need a way to trigger that method. Open **Main.storyboard** and add a button in the middle of the view controller, toward the bottom. Give the button the title "**Play Note**" and change the color of the text to RGB (224, 224, 224). Then wire it up to the `IBAction playNote:` that you added above.

Build and run both the iGuitar app and the AudioHost app. In AudioHost, connect iGuitar and then tap on the **Play Note** button. Like a ghostly Stevie Ray Vaughan, iGuitar plays a solid middle C.

Here's the flow of the events in your apps:

1. AudioHost generates a MIDI event and sends it to iGuitar.
2. iGuitar receives the MIDI event and passes it to its synthesizer audio unit.
3. The synthesizer audio unit generates the audio.
4. The audio flows from iGuitar to AudioHost.
5. If an effect is connected in AudioHost, then audio flows through the effect and back in.
6. AudioHost plays the audio out through the speaker.

You now have three apps that can talk to each other through inter-app audio: the host, an instrument, and an effect.

Challenges

Want to test out your new knowledge of Core Audio? Here’s a challenge for you!

Challenge 1: Selectable effects

Now that you know all about audio graphs and how to swap nodes in and out, you can put that knowledge to good work. iEffects is a bit boring at present as it only has a single effect. Your challenge is to add the ability to select different effects.

To do this, you're going to need a UI to select between different effects. Once you've made that and decided on which effects units you're going to support, write the code to switch between each effects unit.

Tip: You're going to need to write code to unwire and rewire the audio graph as necessary.

Chapter 27: What's New in PassKit, Part 1

By Marin Todorov

The Passbook ecosystem is Apple's technology for distributing tickets, coupons, cards and just about any piece of digital information from your server direct to your customers' iOS devices. Passbook and the PassKit framework were first introduced in iOS 6 and a growing number of companies are adopting Passbook to manage their digital redeemables. With the release of iOS 7, Apple has added a number of features to Passbook such as a new UI, automatic expiration, time zones, and iBeacons.

In this chapter you are going to learn how to take advantage of the new iOS 7 Passbook features, and still remain iOS 6 compatible for those slow upgraders.

Since this book covers only the new iOS 7 APIs you will need to know the basics of creating passes in Passbook and how the PassKit web-service works. You can get up to speed by checking out some or all of the following resources:

- *iOS 6 by Tutorials* (http://www.raywenderlich.com/store/ios-6-by-tutorials) covers everything about creating Passbook passes in over 200 pages of high quality tutorials.
- For just the basics, there's an iOS 6-only version of the Passbook chapter online for free (http://www.raywenderlich.com/20734/beginning-passbook-part-1).
- Apple has its own Passbook documentation and reference material available online (https://developer.apple.com/passbook/).

This chapter is split into two sections. In the first section, you are going to cover the visual updates to passes in iOS 7 and build a checklist for updating your existing passes.

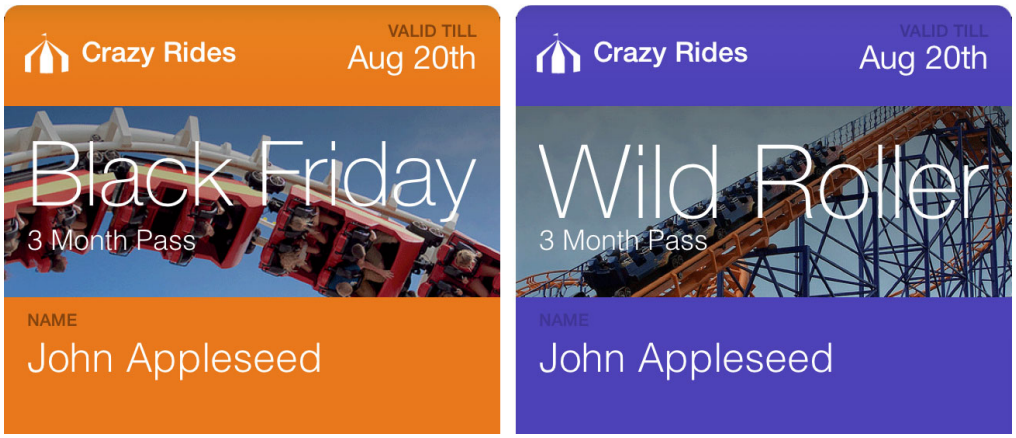
In section 2 you'll work on updating a sample iPhone app and several existing passes provided with this chapter; you are going to fix some existing in iOS 6 issues in the passes and add new features for iOS 7.

Getting started

Throughout this chapter you will be working on a project – the iPhone app of a fictitious amusement park called **Crazy Rides**. For the purposes of this chapter, pretend that the app is live on the App Store and you have a list of issues to fix and several new features to add.

Grab the starter project for this chapter called **CrazyRidesStarterProject**.

The "Crazy Rides" park lets its users buy 3-month passes to its 2 biggest attractions – the "Black Friday" and "Wild Roller":



Furthermore, users can buy a coupon that entitles them to get two cotton candies for the price of one:

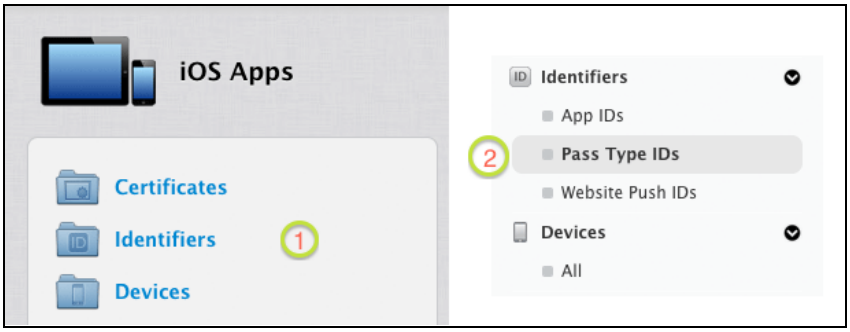
The app is very simple and consists of two view controllers – the home screen where the users buy the different passes, and an extra screen where users can reserve a seat on one of the rides.

> **Note**: The purchasing process in this example app is only a mockup; you can buy as many pretend passes and cotton candy coupons as you like.

Since you will be working mostly on the "Black Friday" pass in this chapter, you will need to do a bit of setup before being able to start working with the code.

Certificate setup

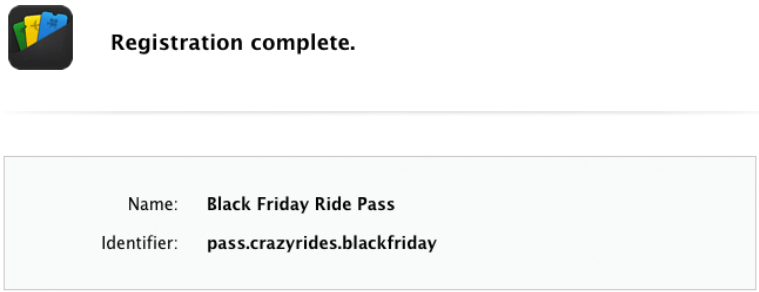
First, you'll need a pass certificate. Sign in to the Apple Developer Center and click on **Certificates, Identifiers & Profiles**. On the next screen, open up the menu item called **Pass Type IDs**, as below:



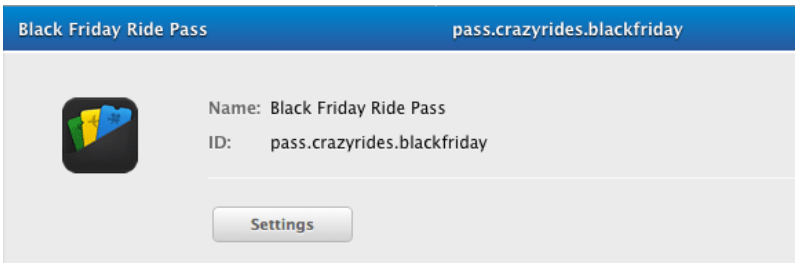
You should now see the list of your existing pass IDs, which might be empty if you haven't created any passes before. Click on the **+** button to add a new pass identifier.

In the next form enter **Black Friday Ride Pass** as the pass **Description** and **pass.com.yourcompany.crazyrides.blackfriday** as the **Identifier**. The IDs need to be unique so fill in your own name or company in the identifier.

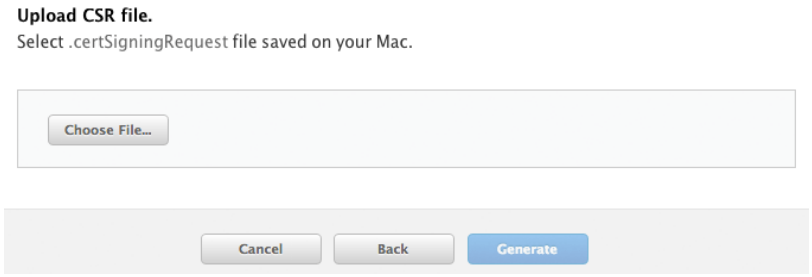
Click **Register** in the Overview scren to finish creating your new pass ID. You will see the following confirmation that the pass was created:



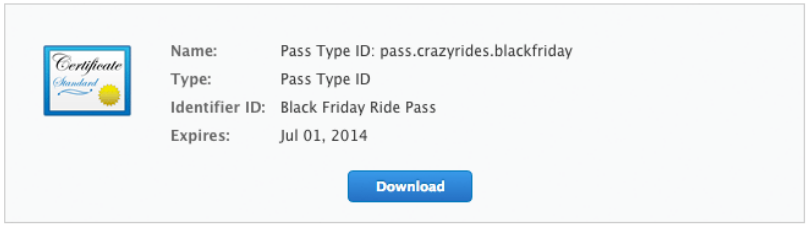
Click **Done** to go back to the pass ID list. Select the **Black Friday Ride Pass** item in the list and click on the **Settings** button, as below:



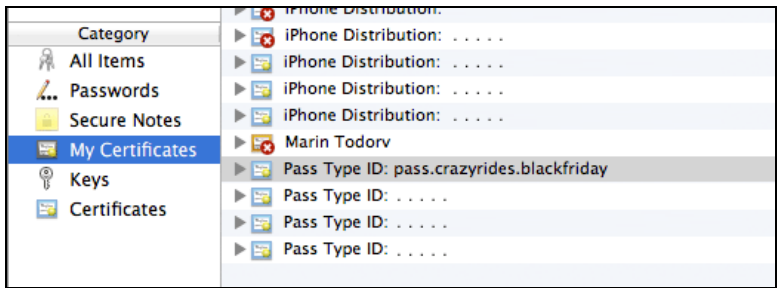
You're now on the certificate generation page. Click **Create Certificate**… On the next screen, you'll need to upload a Certificate Signing Request – just follow the steps listed on the page and click **Continue**. Upload your CSR file and finally click **Generate;** it could take a moment before the next page shows up, so be patient:



You should see the confirmation that the certificate was generated successfully, as so:

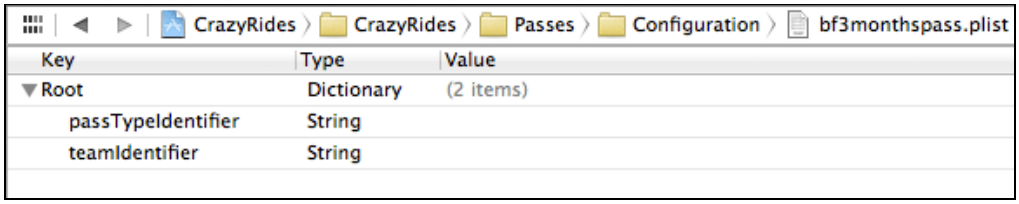


Click on **Download** to transfer the certificate file to your computer. Next, double-click the file to install the certificate in your Keychain. Now you should see the new pass certificate in your certificates list as below:



You might need to select **My Certificates** in the Keychain window to find the new certificate Make sure you can find the certificate with your new pass ID.

Open up the starter project in Xcode; use the Project Navigator to locate and open **CrazyRides/Passes/Configuration/bf3monthspass.plist**. You will see the file contains two keys **passTypeIdentifier** and **teamIdentifier,** as shown below:

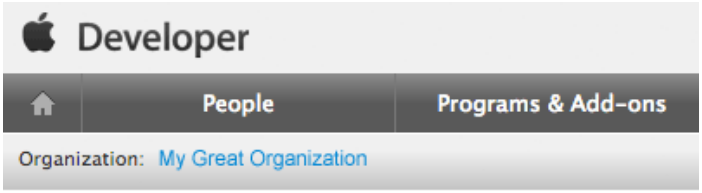


Fill in your unique **passTypeIdentifier** as the value of this key.

If you know your team ID, enter it as value of the other key, skip the instructions below and jump straight to the **Running the Crazy Rides app** section.

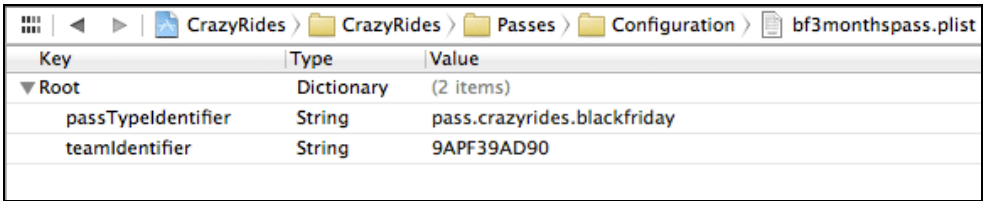
To retrieve your team ID, go to the home page of the Apple Developer Center and click the **Member Center** link at the top of the page.

Towards the top-left corner you will see the name of your organization; click it to retrieve your organization's details.



On the next screen you will see your team ID; it's a 10 character alphanumerical string. Copy that string and paste it into the **teamIdentifier**'s value back in Xcode.

You should have the two configuration variables set, like so:



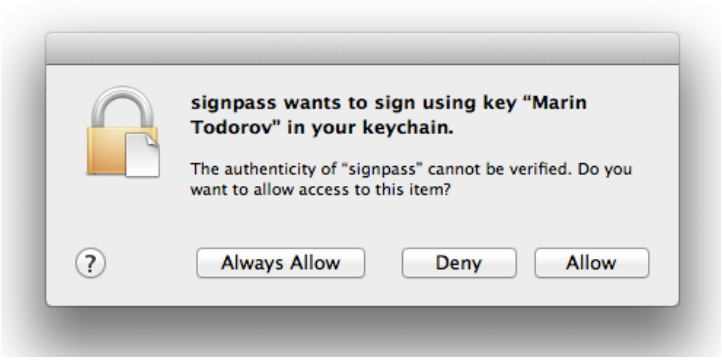
To make your life easier, the starter project includes a pass-building script that rebuilds the Black Friday pass each time you build the Xcode project. This way you can focus on working on the pass code and leave the building of manifests and signature creation to Xcode.

Note that you may need to make the pass-building script executable on your machine before continuing. To do this, make sure the scripts are executable by using the `chmod 755` command.

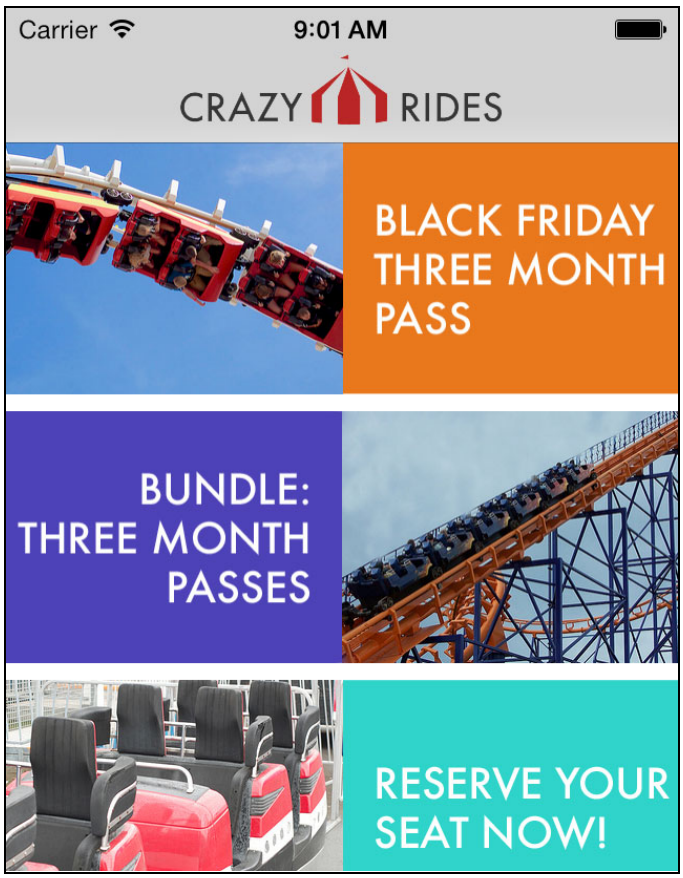
Running the Crazy Rides app

Build and run your app; if a dialogue pops up and asks you to give Xcode access to your Keychain, that's to be expected. Click **Allow** or **Always Allow** if you don't want to be asked every time you run the project**.**

> **Note:** Xcode runs an app called `signpass` to generate the passes that go into the app's bundle. You'll explore that in more detail in just a bit.



The app will then appear in all of its loop-the-loop glory:



Everything appears to be hooked up correctly; you can move on to working with the code.

Note: For those of you who are curious to know what's happening during the build process, here's what Xcode does behind the scenes.

First, Xcode runs **buildpasses.sh** in the project directory before compiling the app. **buildpasses.sh** executes `passdata`, which grabs the values from

bf3monthspass.plist and puts them into **pass.json**. The source code of `passdata` is provided along with the rest of this chapter's assets.

`signpass` grabs the pass source files and compiles them using the matching certificate from your keychain. The source code for `signpass` is available from Apple here: https://developer.apple.com/downloads/index.action?name=Passbook)

Finally, Xcode includes **bf3monthspass.pkpass** in the app bundle and runs the app.

Visual updates to passes in iOS 7

Before starting to dig into the code, consider what the iOS 7 visual update means for Passbook. Have a look at the same pass previewed in Passbook on iOS 6 on the left, and iOS 7 on the right:

Notice the updates to the pass to bring it in line with the new iOS 7 design paradigm:

- The text is now flat rather than inset.
- All uppercase letters are used for smaller sized labels.
- No gradient is applied to the front of the pass.
- No shine is applied to the pass picture, even if you specify you want it.
- The buttons match the iOS 7 UI.

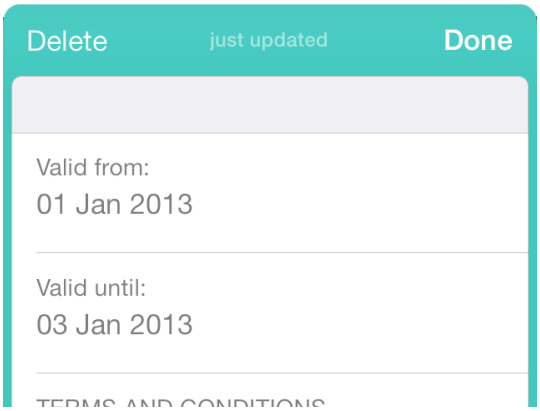
- The width of the pass is now the full width of the screen: 320pt.
- Finally, you have an extra button to share this pass with others.

Due to the new styling, the layout of the fields might vary between iOS 6 and iOS 7 for some passes. The changes aren't especially jarring, but there are some quick steps you can take to ensure iOS 7 compatibility:

1. Open the pass in iOS 7 and check that the layout is preserved.
2. The fonts and UI are automatically changed by Passbook; still, it's up to you to check whether the pass still looks okay.
3. Your strip image is scaled up to match the new 320pt width, but you should update the original image to match the new pass size for pixel perfection.
4. Remove any etching you had previously applied to your logo image.

These tweaks should be enough for 95% of the passes you'll encounter. As with general app design, your passes should fit with the new iOS 7 style by removing extra shine, shadows, etching or bevels on images.

Don't worry about the back of your existing passes; visually, the back remains largely the same as before, except the buttons at the top are now iOS 7 style buttons, as shown below:



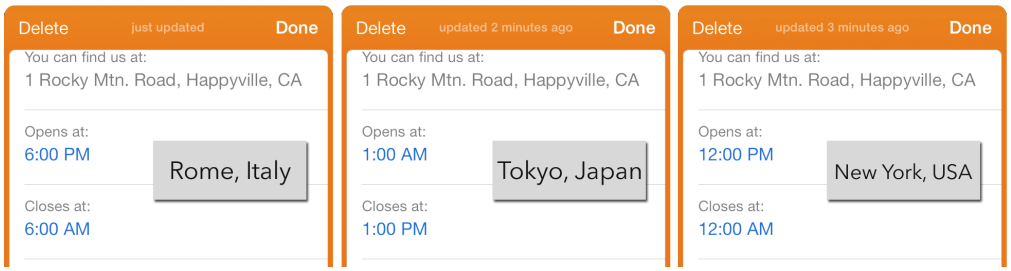
Fixing iOS 6 issues

The existing Crazy Rides has a list of pass issues for you to fix. Your task is to work through this list and fix the issues one by one for all of your iOS 7 customers!

Wrong time zone offsets for date fields

Run the starter project and open the Black Friday pass. Flip it over by tapping the **i** button and have a look at the opening and closing times. The amusement park is open from 8 AM until 8PM, but Passbook automatically adjusts the date fields to your own time zone.

Depending on your location you'll see different times, but here's what the back of the pass looks like in some cities:



To prevent this kind of confusion, Apple introduced a new key for the date fields called `ignoresTimeZone`. In the starter project directory, navigate to the file **passes/bf3monthspass/pass.json** and open it with your favorite text editor.

Scroll through the file and find where the two date fields are located. They will look like this:

```
{
  "dateStyle" : "PKDateStyleNone",
  "label" : "Opens at:",
  "key" : "date",
  "value" : "2013-08-19T08:00-02:00",
  "timeStyle" : "PKDateStyleShort"
},
{
  "dateStyle" : "PKDateStyleNone",
  "label" : "Closes at:",
  "key" : "dateEnd",
  "value" : "2013-08-19T20:00-02:00",
  "timeStyle" : "PKDateStyleShort"
}
```

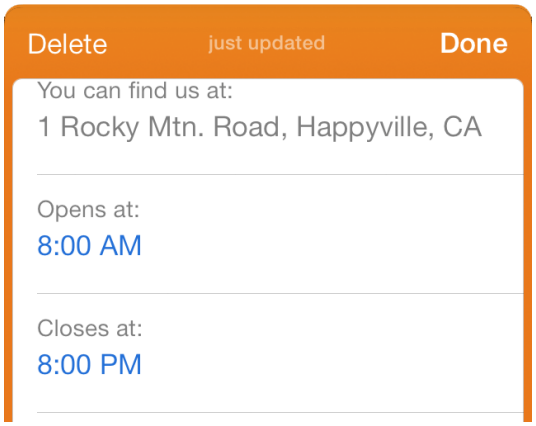
Under each opening curly brace, add the following key to tell Passbook to ignore the time zone offset:

```
"ignoresTimeZone" : true,
```

Don't forget the comma at the end of the line!

That's everything you need to do about the time zone issue.

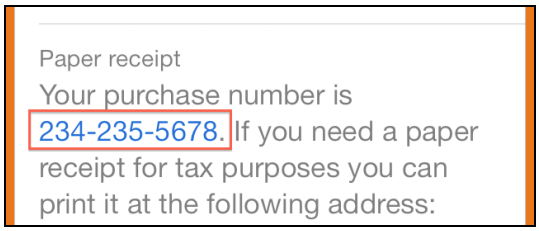
Build and run your project, open the Black Friday pass, and flip it over: you can see the correct opening and closing times no matter where you're located:



Data-detector false positives

Data detectors are great when you use them in your own apps; however, in Passbook the back fields have *all* data detectors enabled for *all* the text *all* the time, so sometimes pieces of text are recognized in error.

You might have noticed the following false positive on the back of the Black Friday pass:



The text clearly states "Your purchase number is …" but since that particular purchase number is also a valid telephone number, Passbook links that number to the Phone app.

In iOS 7 you can specify which data detectors should be enabled. Open **passes/bf3monthspass/pass.json** in your text editor again and find the **receipt** field as follows:

```
{
  "label" : "Paper receipt",
  "key" : "receipt",
  "value" : "Your purchase number is 234-235-5678. If …"
}
```

`dataDetectorTypes` is the new key that controls which data detectors are enabled; by default, all detectors are enabled. You can use one or a combination of detectors from the following list:

- `PKDataDetectorTypePhoneNumber`: phone numbers
- `PKDataDetectorTypeLink`: web URLs

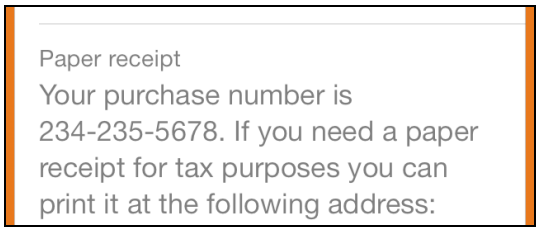
- `PKDataDetectorTypeAddress`: addresses
- `PKDataDetectorTypeCalendarEvent`: dates, time ranges, etc.

The receipt field has two pieces of interesting information for data detectors: the purchase number (which looks like a phone number) and a URL to a support page. You want the URL data detector to work, but you need to disable that pesky phone number data detector.

Before the opening curly bracket in that last code fragment, add the following line:

```
"dataDetectorTypes" : [
  "PKDataDetectorTypeLink"
],
```

Now run the app again and you'll notice that the purchase number is not recognized as a phone number anymore:



Making links easier on the eyes

If you read through the whole text in the **receipt** field on the back of the pass, you'll see that the owner of the pass can tap on the provided link and print themselves a paper receipt. Again, all of this functionality is just mocked up this sample project.

The included link takes up a lot of space and isn't all that attractive:

```
www.crazyridesllc.com/receipts/print.php?show=clientconfirmation&r
eceiptId=234-235-5678&source=passbook
```

Apple adds a new key **attributedValue** as an alternative to the **value** key; it can contain a restricted set of HTML markup to provide a text link instead of the raw URL using the **<a href>** tag.

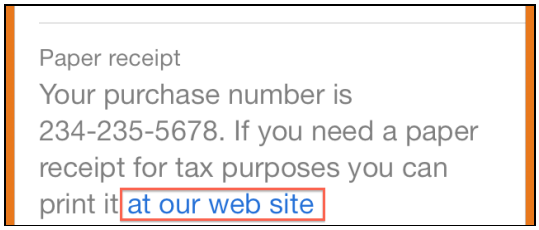
Add this key to the **receipt** field above the existing `value` key:

```
"attributedValue" : "Your purchase number is 234-235-5678. If you
need a paper receipt for tax purposes you can print it <a
href='http://www.crazyridesllc.com/receipts/print.php?show=clientc
onfirmation&receiptId=234-235-5678&source=passbook'>at our web
site</a>",
```

The contents of the `attributedValue` are similar to those of `value`; the only difference is that the link is wrapped up in an `<a>` tag to look a little more attractive when it's rendered.

You still need to leave the `value` key for the users running on iOS 6 – they're stuck looking at the ugly link until they upgrade to iOS 7.

Build and run your app, and check out the **Paper receipt** field, which looks much nicer now:



To wrap up the list of potential iOS 6 Passbook migration issues, iOS 7 will now check that fields on the back of the pass have unique names. Some passes that work on iOS 6 will be invalid when installed on iOS 7.

Passbook improvement in iOS 7

Now you are going to add a few features to your pass that are only available in iOS 7.

Handling expiration dates

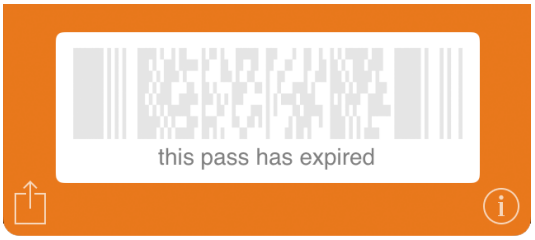
If you look at the header on the front of the pass, you will notice that the pass has an expiration date. However, when this expiration date comes and goes, nothing really happens. Your server needs to send an update to the pass and somehow turn it into an expired pass.

In iOS 7 there's a new pass field to specify the expiry date of the pass. Even if the user is not online, Passbook will display the pass as expired. The new pass JSON key is called `expirationDate` and accepts a standard W3C formatted date.

The expiration date on the Black Friday pass header is the 20th of August, so add a new JSON key to the top level in **passes/bf3monthspass/pass.json** as follows:

```
"expirationDate" : "2013-08-20T00:00-02:00",
```

Build and run your app; you should see the barcode dimmed and an expiration message displayed, as below:



It's important to notice that the pass is not automatically destroyed; it remains in the user's pass library and the user can still interact with it.

Just as with iOS 6, it's up to your backend service and database logic to actually expire the passes. The nice expiration message is really a convenience method to show the customer that the pass is expired.

Open up **pass.json** and change the expiration date to the following:

```
"expirationDate" : "2020-08-20T00:00-02:00",
```

This way you can keep working with the pass, as it will be good well into the year 2020! If it's already 2020 and you're reading this (and still working with iOS 7), we need to have a little chat.

Immediate invalidation

Sometimes you want to expire a pass immediately. There can be many reasons for this: the payment for the pass was denied, the user cancelled the purchase manually, or maybe a user had couple of beers too many and is now banned from the amusement park!

Doing this on the backend side is usually pretty easy – just delete or hide the pass in the database. But how do you notify users that their pass is not valid anymore?

In iOS 7 there's a new way to do that. Add the following code to the top level of your JSON in **passes/bf3monthspass/pass.json:**

```
"voided": true,
```

When you update the pass with this field it will immediately show the user that it's expired.

If you plan to use the `expirationDate` or `voided` fields in your pass, it's a good idea to add these expiration policies to your terms and conditions right on the back of the pass itself.

Remove the `voided` line you added to the JSON; you'll need a valid pass to work with the rest of the examples in this chapter and the next.

Fine tune location relevancy

One of the best features of Passbook is that when you're close to a location that accepts passes that you own, your lock screen will show any relevant passes to remind you to use them.

Assume Crazy Rides wants the Black Friday pass to appear on the device's lock screen only when the user is within 15 meters of the ride's entrance. You would add the relevant location information with the new `maxDistance` key, like so:

```
"locations": [
  {
    "latitude"   : 42.136848,
    "longitude"  : 24.73196,
    "maxDistance": 15
  }
],
```

That's all you need to do to refine the location relevance of the pass. Actually you can be even more precise by using iBeacons, which you'll implement in the next chapter.

Challenges

You've already covered all the basics of the new in iOS 7 Passbook features. You will continue with the new Passbook APIs in the next chapter by learning how to implement a whole new level of location awareness to a pass, working with pass bundles, and increasing the connection between passes and apps.

However you can still achieve an even better result on the Black Friday by taking on a challenge by yourself.

Challenge 1: expiration date backfield

Add a new date field to the Black Friday pass. Set it to display the month, day and year, make it ignore the time zone offset of the current user location.

If in doubt check the pass backfields specification and the date field properties online:

https://developer.apple.com/library/ios/ -documentation/UserExperience/Reference/PassKit_Bundle/Chapters/FieldDictionary.html

Chapter 28: What's New in PassKit, Part 2

By Marin Todorov

In the previous chapter you worked on the Black Friday pass that comes with the Crazy Rides app. You made a number of improvements to the pass, making use of the new pass features available in iOS 7.

However, you didn't get to look at what's new with the **PassKit.framework,** and you definitely haven't seen some of the really cool stuff like working with iBeacons and distribution via QR codes.

This chapter will take you through the rest of the Passbook updates in iOS 7. You'll continue to work with the Crazy Rides app and its integration with Passbook.

Note that you need two test devices running iOS 7 to test the iBeacon portion of this chapter.

Pushing Passbook integration further

In this section of the book you are going to take on more complicated requests from your fictional employer Crazy Rides and provide much better Passbook integration for their business.

Using iBeacons to activate passes

In the previous chapter you set the Black Friday location relevancy precision to make the pass appear when the user was a certain distance from the ride entrance. But imagine the following scenario: a shuttle train runs around the Crazy Rides amusement park and takes visitors directly to the Black Friday ride. Passengers need to present their Black Friday pass to get on the train.

The pass needs to know when it's in the vicinity of the current location of the train, but the train's a moving target! You can't hard code the coordinates of a moving train into your app. However – this seems like a perfect opportunity for an iBeacon!

> **Note**: If you're new to iBeacons, check out Chapter 24, "What's New in Core Location" for tons of background info and a fun example.

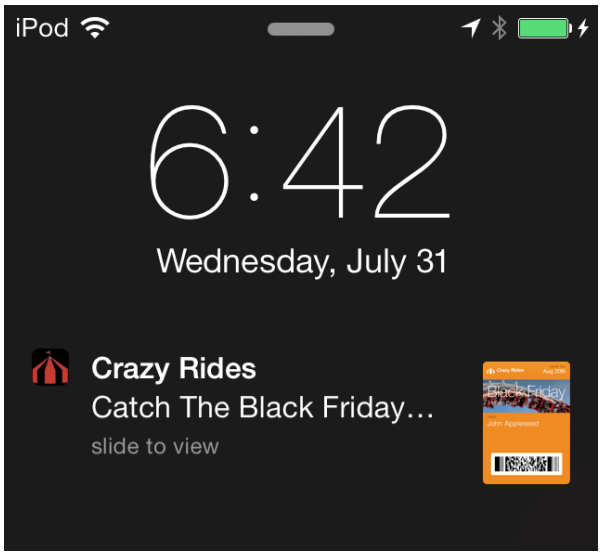
If the train had an iBeacon on board, you could instruct Passbook to show the pass on the lock screen in the vicinity of that beacon.

Locate **passes/bf3monthspass/pass.json** and open it up in your favorite text editor. Add the following key to the top level of the JSON code:

```
"beacons" : [
  {
    "proximityUUID" : "74278BDA-B644-4520-8F0C-720EAF059935",
    "relevantText" : "Catch The Black Friday Train!"
  }
],
```

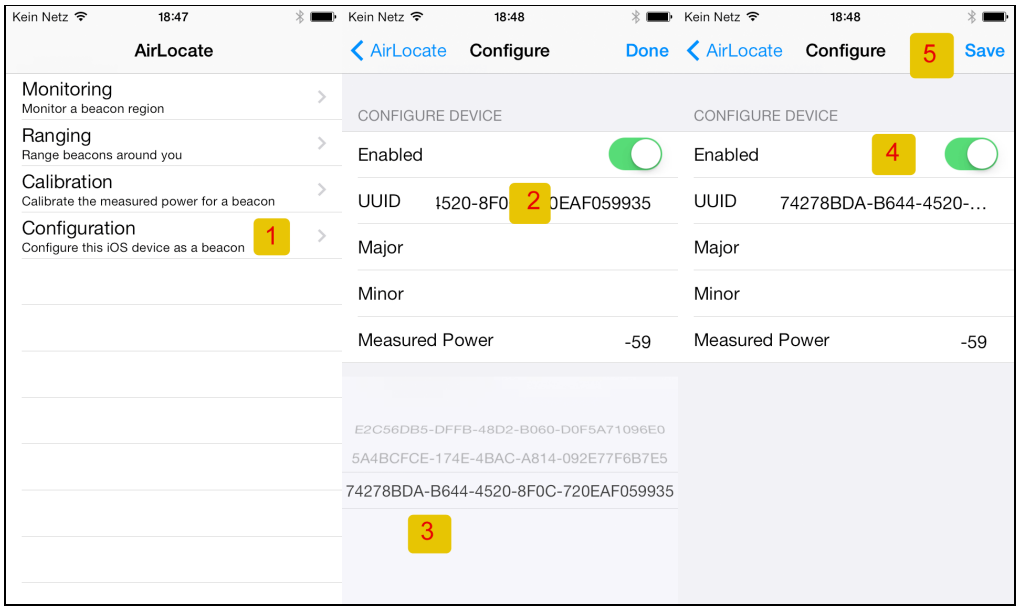
The `proximityUUID` key specifies the UUID of the Bluetooth beacon, and `relevantText` contains the lock screen notification message. Since beacon region monitoring is now a built-in feature in iOS 7, Passbook only needs the UUID and will detect the beacon automatically when it's nearby.

When you come near a beacon, the pass notification will appear on your lock screen, as shown below:



Note the preview of the actual pass – another nice feature of iOS 7.

To test this feature, you will need an iOS 7 device to act as your Bluetooth beacon. Apple has an iBeacon test app called **AirLocate**. Open https://developer.apple.com/downloads/index.action and search for "AirLocate" to find the sample app. Install the AirLocate app, and perform the following configuration on your device as shown below:



1. Tap on **Configuration**.
2. Tap **UUID** to change the beacon UUID to match what you entered in the JSON earlier. The app ships with a few pre-defined UUIDs, and it just so happens that the third entry in the list matches up exactly to what you entered in the JSON earlier, so just select that!
3. Turn on the **Enabled** switch.
4. Tap on **Save** to persist your changes.

Once your changes are saved, your device will become an iBeacon and will keep broadcasting the UUID you selected as long as the app is in the foreground. The lock screen sends the app to the background and kills the iBeacon broadcasting, so keep an eye on it.

Have a look at your primary device; once it picks up on the iBeacon from your secondary device, you should see a notification appear on your lock screen for the pas in question.

Pass bundles

Now it's time to finally have a look into improving the iPhone app itself.

Open **MasterViewController.m** and find the following line at the top of the file:

```
#import <PassKit/PassKit.h>
```

Replace this with the new `@import` syntax as below:

```
@import PassKit;
```

Often your customers will want to purchase several passes at once; until now the user had to view and download one pass at a time, which was a tedious operation. You can now offer pass bundles, where users can buy one package containing all the passes.

If you scroll down the file and take a look at the following method:

```
-(PKPass*)passWithName:(NSString*)name
```

This method loads the contents of a .pkpass file from the app's bundle and initializes a new `PKPass` instance with the contents of the pass file. A bit further down, you'll find the next method:

```
-(void)showPassWithName:(NSString*)name
```

This method calls `passWithName:` to get a `PKPass` instance and uses a `PKAddPassesViewController` to display the pass on the screen which gives the user a chance to add this new pass to their Passbook library.

Your task is to build upon this code so users can get a bundle of passes in one shot. First, you are going to create a simple method to create an array of all the passes that'll make up the pass bundle. Add the following method to the `MasterViewController` class body:

```
-(NSArray*)passBundle
{
  return @[[self passWithName:@"bf3monthspass"],
           [self passWithName:@"ww3monthspass"],
           [self passWithName:@"cottoncandy2for1"]];
}
```

This simple method returns an array containing the three passes bundled in the app.

You need one more method to present these passes to the user. Add the following method:

```
-(void)showPassBundle
{
//show the 3 passes in the add pass controller
NSArray* passes = [self passBundle];

PKAddPassesViewController* addPasses =
  [[PKAddPassesViewController alloc] initWithPasses: passes];

[self presentViewController:addPasses
                   animated:YES
                 completion:nil];
```

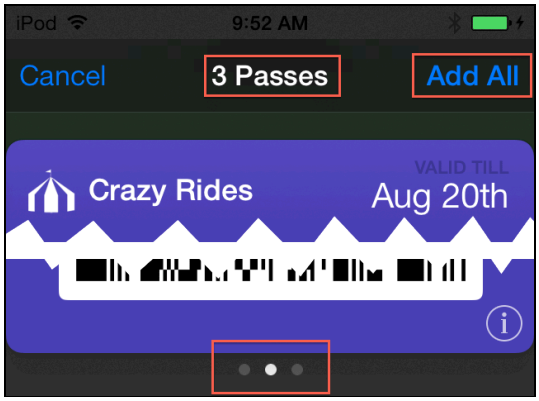
```
}
```

Here you create an instance of `PKAddPassesViewController`, but this time you use the new in iOS 7 initializer `initWithPasses:` that takes an array of `PKPass` objects as opposed to a single pass.

Now find to the method stub for `actionPassBundle:` and add the following line:

```
[self showPassBundle];
```

That's it! Build and run your app and tap the "Bundle: three month passes" option to try the new functionality you've just built:



Tap the **Add All** button at the top of the pass display to add all three passes to your library at once. That's much easier than adding them one at a time.

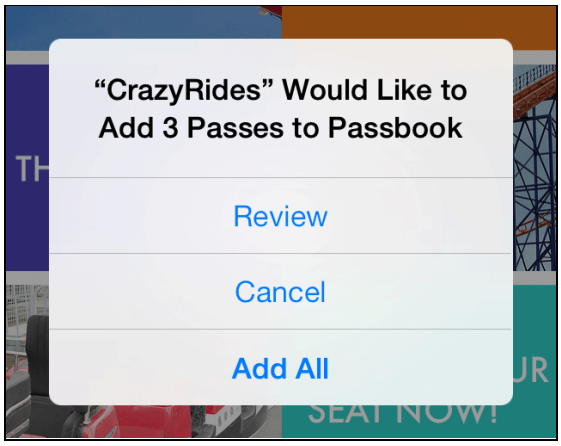
But what if Crazy Rides expanded to a hundred rides – would users even want to have a look at the pass previews? You'll need to offer users the chance to add all the available passes without previewing them first.

Replace the contents of `actionPassBundle:` with the following code:

```
//fetch the user pass library
PKPassLibrary* passLibrary = [[PKPassLibrary alloc] init];

//add 3 passes at once
[passLibrary addPasses:[self passBundle]
 withCompletionHandler:^(PKPassLibraryAddPassesStatus status) {
         //callback block

    }];
```

With this new approach, you fetch an instance of `PKPassLibrary` and call `addPasses:withCompletionHandler:`, passing the list of passes and a callback block. This API will attempt to install the passes without displaying them on screen.

Build and run your app; tap the button to add the passes to your library. You'll be presented with the following alert message:



Cancel and Add All will work as expected, but if you tap on **Review** nothing happens. This is because it's up to you to present the bundle for review to the user. Luckily, you have this code already — you just need to hook it up.

The callback block takes a `status` parameter; this status code of type `PKPassLibraryAddPassesStatus` is your clue to whether the user wants to review the passes or have them added straight to their library.

The `PKPassLibraryAddPassesStatus` enumeration contains three constants:

1. `PKPassLibraryShouldReviewPasses` – the user wants to review the passes and it's up to you to present them on screen.
2. `PKPassLibraryDidCancelAddPasses` – the user cancelled the operation.
3. `PKPassLibraryDidAddPasses` – the user added the passes to their library.

Paste the following code in the empty completion block body:

```
//call on the main thread
[self performSelectorOnMainThread:
      @selector(bundleAddDidCompleteWithCode:)
      withObject:@(status)
      waitUntilDone:NO];
```

There is no guarantee as to which thread the callback block will end up on, so you need to switch to the main thread to interact with the UI.

Next, add the following method:

```
-(void)bundleAddDidCompleteWithCode:(NSNumber*)status
{
//action completed
if ([status intValue]==PKPassLibraryShouldReviewPasses) {
```

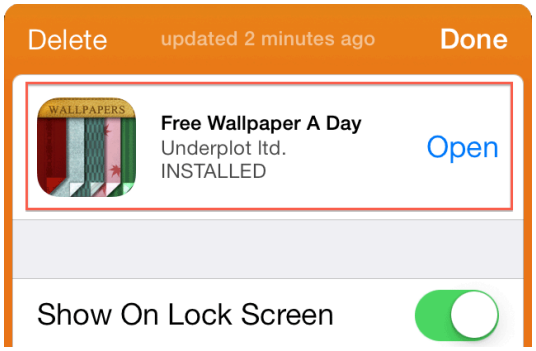
```
  //the user wants to review the pass bundle
   [self showPassBundle];

} else if ([status intValue]==PKPassLibraryDidAddPasses) {
  //success message
  [[[UIAlertView alloc] initWithTitle:@"Thanks!"
    message:@"Thanks for purchasing the 2 passes bundle, we've
thrown in also a cotton candy coupon. Enjoy!"
    delegate:nil
    cancelButtonTitle:@"Close"
    otherButtonTitles: nil] show];
 }
}
```

If the status code is `PKPassLibraryShouldReviewPasses`, then you fire up the familiar `showPassBundle`; if the user added all passes, show a confirmation alert.

Build and run your app; this time tapping the **Review** button displays the pass bundle where the user can review each of the passes before adding them to the library.

Connecting to your app on a different level

One of the best Passbook features is that the user can jump to your app directly from one of their passes in Passbook. You've probably already seen app icons on the back of different passes; when Open is tapped, the target app launches on the device:



You achieve this by including the relevant app IDs to your pass app store identifiers, like so:

```
"associatedStoreIdentifiers" : [
  403497940, 424600440
],
```

You can include multiple IDs here, and Passbook only uses the first one it encounters in `associatedStoreIdentifiers` that matches the device type. For instance, you can include the app IDs of an iPhone-only app and an iPad app, and Passbook will automatically select the ID appropriate for the current device.

Launching the related application is convenient, but in iOS 7 you can now send additional data from the pass to the application. That means you can add some special behavior to your app when it launches via Passbook.

Acknowledgment

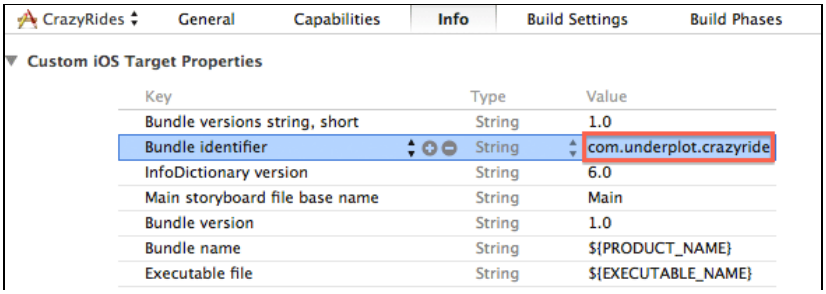
Since the connection between a pass and an app is based on the App Store ID, you have probably guessed you will need a live app in the App Store to implement this next part. If you have any apps at all in the App Store, you can use their app and bundle IDs. However, if you don't have any published apps, you can follow along by reading, or skip straight to the **Distributing passes with QR codes** section.

This is indeed a hack, but since Passbook doesn't provide a Sandbox service for proper testing, this is the way you will have to do it. This testing will only affect your test device and will not affect the real life users of your app.

First, delete the Crazy Rides app from your device. Then install your own app from the App Store on your device.

Next, find your app's Bundle ID. The easiest way to do this is to go to your app's **Info.plist** and grab the ID which follows the format **com.yourdomain.yourappname.**

Switch back to the Crazy Rides app in Xcode and set your bundle ID as the bundle ID of Crazy Rides, as so:



Press and hold the **Option** key on your keyboard and choose **Product/Clean Build Folder ...** to clean the project build location. This will ensure any new builds use the new bundle ID.

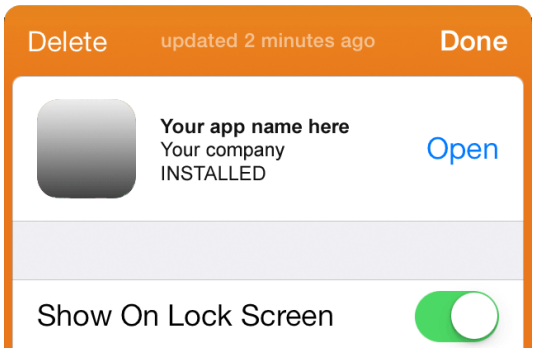
To fetch your app's App Store ID, head to https://itunesconnect.apple.com and log in with your Apple credentials. Click on the link **Manage Your Apps**. Select the app you will be using for this chapter, and on the next screen you'll find your numeric **Apple ID**.

Now open up **passes/bf3monthspass/pass.json** in a text editor and add your Apple ID to the top level of the JSON hierarchy:

```
  "associatedStoreIdentifiers" : [
    0000000000
  ],
```

Fill in your own App Store ID for 0000000000. Don't add quotes around the value since it's a number, not a string.

Now launch the Crazy Rides app. You'll notice it replaces the real app on your device with Crazy Rides since the bundle IDs match. Add the Black Friday pass to your Passbook; flip the pass over and you'll see the associated app strip:



Tapping on this button should open the Crazy Rides app.

One way to improve the user experience with Black Friday is to take the user directly to the "reserve a seat" screen when somebody launches the Crazy Rides app from the Black Friday pass.

Switch to the JSON source file of the pass and add the following key:

```
"appLaunchURL" : "crazyrides://bookseat",
```

When you specify an `appLaunchURL` for the pass, Passbook will not only launch your companion app, but will also pass this URL to the app.

First you'll add the URL scheme **crazyrides** to your app. Select the **Crazy Rides** project file at the top of the Project Navigator in Xcode, then make sure that the **Info** tab is selected. Scroll down and open the strip saying **URL Types (0)**. Click on the **+** button to add a new URL scheme to the app.

Enter **passes.crazyrides** as the **Identifier**, and **crazyrides** as the **URL Schemes**. That's all you have to do; your form should look like the image below:

Select **Product/Clean** from Xcode's menu and build and run your app so the URL scheme is registered. Your app is now registered to handle URLs for the **crazyrides** scheme; however, your app won't do anything specific when the app is invoked with a URL.

To add this functionality, open **AppDelegate.m** and add the following stub:

```
/* if app is launched from a pass, go to booking screen */
-(void)handleURL:(NSURL*)url userInfo:(NSDictionary*)info
{

}
```

You'll come back to this in just a second. Find `application:didFinishLaunchingWithOptions:`, and add the following code just before the final `return` statement:

```
/* handle app launch from URL */
if (launchOptions[UIApplicationLaunchOptionsURLKey]) {

[self handleURL: launchOptions[UIApplicationLaunchOptionsURLKey]
 userInfo:launchOptions[UIApplicationLaunchOptionsAnnotationKey]
];

}
```

If a URL was passed to your app, you'll find it in the `UIApplicationLaunchOptionsURLKey` key of the `launchOptions` dictionary. The `UIApplicationLaunchOptionsAnnotationKey` contains any additional data objects that were passed to the app at launch time.

The code above covers the case when your app isn't running and is launched by tapping on the button on the back of the pass. Additionally, the `UIApplicationDelegate` protocol features a separate method to handle situations when your app is already running and invoked from a URL.

Add the following method to the `AppDelegate` class body:

```
- (BOOL)application:(UIApplication *)application
  openURL:(NSURL *)url sourceApplication:(NSString *)
  sourceApplication annotation:(id)annotation
{
    /* if app is awakend from a pass, go to booking screen */
    [self handleURL: url userInfo: annotation];
    return YES;
}
```

In the method above, you simply pass the URL and the annotation object along to your own method.

Now you can implement the body of `handleURL:userInfo:`. It needs to handle two distinct situations; one where the app has just started, and another where the app is already running and is could have any view opened.

Replace the body of `handleURL:userInfo:` with the following code:

```
-(void)handleURL:(NSURL*)url userInfo:(NSDictionary*)info
{
  if ([url.host isEqualToString:@"bookseat"]) {
    //open the book ride screen, and let it handle the booking
    UIApplication* app = [UIApplication sharedApplication];
    UINavigationController* navigationCtr=
      (UINavigationController*)app.keyWindow.rootViewController;
    [navigationCtr popToRootViewControllerAnimated:NO];
    [navigationCtr.topViewController
    performSegueWithIdentifier:@"bookRide" sender:nil];
  }
}
```

First you check if the `host` property of the `url` equals `bookseat`; if so, then show the screen to reserve a seat. Next, you get a handle to the app's navigation controller and pop all view controllers except the home screen. Then all you need to do is perform the `bookRide` segue, which pushes the reserve seat view controller to the view controller stack.

Build and run your app and add the Black Friday pass to your Passbook again. Now switch to Passbook and flip over the pass; you should see the button to the companion app. Tap **Open** and … oh joy! Crazy Rides opens up takes you straight to the reserve a seat screen, as below:

Wouldn't it be great, though, if the user could tap **Open** on the Black Friday pass and see the reservation dialogue for that particular ride?

You can do that by adding additional information to the `appLaunchURL` key.

Open up **passes/bf3monthspass/pass.json** and modify the `appLaunchURL` as below:

```
"appLaunchURL" : "crazyrides://bookseat/?Black%20Friday",
```

You've added a *query string* to the URL containing the text "Black Friday". It's been URL encoded, so the space between the words is represented by `%20`.

> **Note:** If you want to read more about URLs or you are unclear about what a query string is, read up on it here:
>
> http://en.wikipedia.org/wiki/Uniform_resource_locator

Now that the extra query string information is coming along for the ride, you need to do something with it. Open **AppDelegate.h**, and add the following property just after the `window` property:

```
@property (strong, nonatomic) NSString* bookingSeatOnRide;
```

This property holds the name of the ride that is being sent through `userInfo`.

Now open **AppDelegate.m** and find the following line of code:

```
if ([url.host isEqualToString:@"bookseat"]) {
```

Add the following code directly underneath:

```
self.bookingSeatOnRide = [url.query
  stringByReplacingPercentEscapesUsingEncoding:
  NSUTF8StringEncoding];
```

This fetches the incoming query string and calls `stringByReplacingPercentEscapesUsingEncoding:` to decode the special characters, such as `%20` for the space character. It then saves the decoded string to the new class property.

Now it's finally time to look into the code of `BookSeatViewController`, which is where the user can reserve a seat on a ride.

Open **ViewControllers/BookSeatViewController.m**. A quick look reveals some key methods:

```
-(IBAction)actionReserveSeat:(UIButton*)sender
```

This action method is connected to both buttons – it checks the `sender`'s tag and determines the name of the ride. Then it passes this ride name to the following method:

```
-(void)bookRideWithName:(NSString*)rideName
```

This method takes a ride name and shows a confirmation dialog to the user where they have the option to proceed with the reservation or cancel it altogether.

```
-(void)alertView:(UIAlertView *)alertView
didDismissWithButtonIndex:(NSInteger)buttonIndex
```

This is the `UIAlertView` callback method – if the user taps the button **Absolutely** – they will receive a confirmation for their reservation.

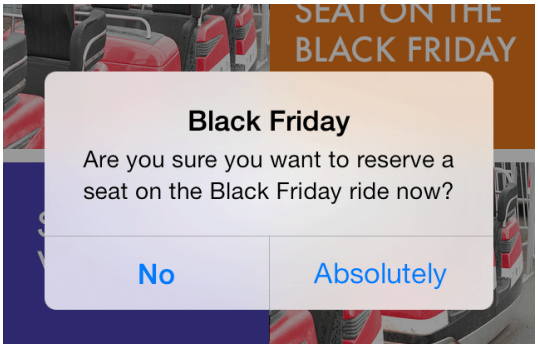
Your task is to add some code to `viewDidAppear:` to check if a ride name was passed in from a URL and stored in the `AppDelegate`'s property. If so, it invokes `bookRideWithName:` right away.

Add the following code to `viewDidAppear:`

```
AppDelegate* ad = [UIApplication sharedApplication].delegate;if
(ad.bookingSeatOnRide) {
  [self bookRideWithName: ad.bookingSeatOnRide];
  ad.bookingSeatOnRide = nil;
}
```

If the app delegate has something stored in `bookingSeatOnRide`, pass it on to `bookRideWithName:`. You then set `bookingSeatOnRide` to `nil` so that the dialog doesn't show up automatically the next time this view controller appears on the screen.

Build and run your app; this time when you click on the App icon on the back of the pass, you'll be asked right away to confirm your seat reservation on the Black Friday ride:



What other information can you send from Passbook to your app, and how will you use it? That's only limited by your imagination!

Distributing passes with QR codes

The final feature you'll cover in this chapter is distributing passes via QR codes. The iOS 7 SDK now includes a QR code reader, which is integrated into Passbook.

> **Note**: Curious about the new machine-readable code detectors included in iOS 7? Check out Chapter 22, "What's New in AVFoundation" for more details.

The Crazy Rides people are very happy with their iPhone app and the passes they distribute to customers. They see a fair amount of digital pass usage, which is making their lives considerably easier.

They now want to conduct a survey in the park so their visitors can rate their experience on the Black Friday ride, and they would also like to monitor the results of the survey in real-time.

To do this, you will make use of the new built-in QR scanner in Passbook. If you haven't used it yet, launch Passbook right now and tap **Scan Code**. The scanner will read a QR code with a URL that points to a .pkpass file.

This is great for distributing coupons and passes, but how does this apply to a survey? With a backend web server, you can prepare two QR codes which point to the following URLs:

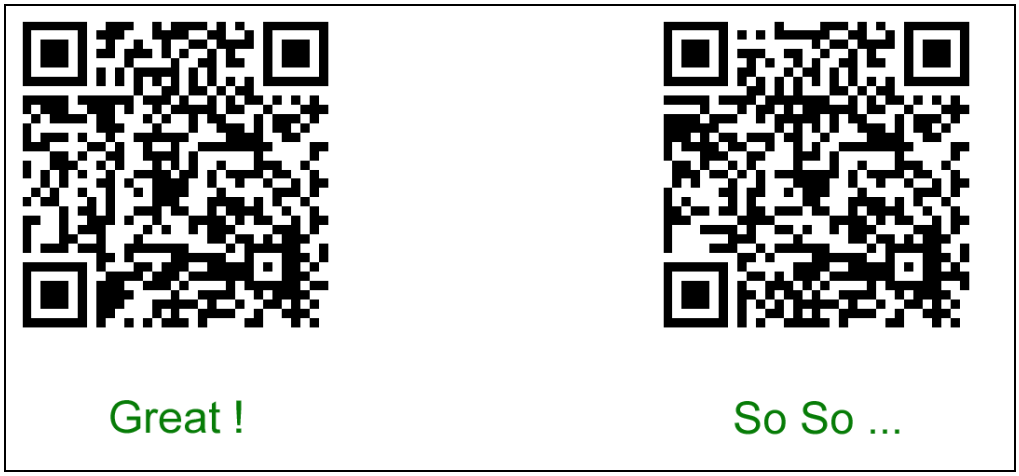
```
https://yourdomain.com/getPass.php?answer=Great&source=rideExit
https://yourdomain.com/getPass.php?answer=SoSo&source=rideExit
```

As you see, both URLs point to the same PHP script, but one of the codes passes the survey answer "Great" and the other "SoSo".

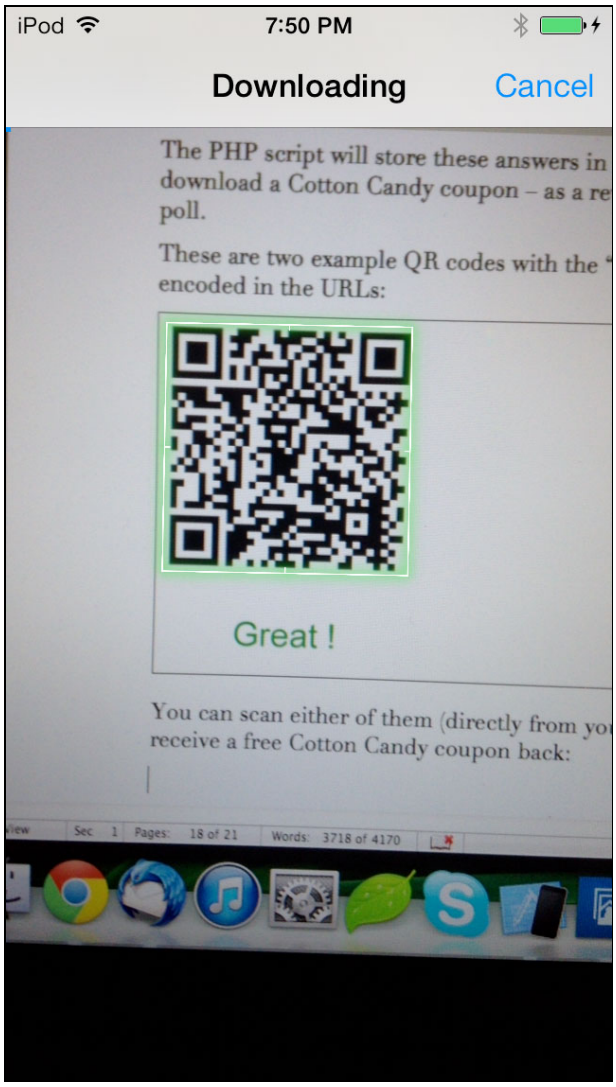
> **Note**: Getting a web server with PHP set up is out of scope for this chapter, but you can at least follow along with the concepts. If you have the server

skills, a sample implementation of **getPass.php** is included in the resources for this chapter. It writes out the survey responses to a CSV file and returns a Cotton Candy coupon as a reward to the people taking the poll.

These are two example QR codes with the "Great" and "So So" answers encoded in the URLs:



You can scan either of them, and a Candy Coupon will start downloading to your Passbook, as so:



As the actual connection to your website and the pass download happens on the customer's iPhone, you don't need to do anything else besides display the QR codes.

For more details on how the poll works behind the scenes, have a look at the folder **blackFridayPoll** in this chapter's assets. The 20 lines of PHP are pretty straightforward; they simply store the results in a .csv file and serve up a pre-built .pkpass files for download.

Feel free to experiment with this; you can dynamically build different passes and provide them for download, maybe include a personal message and/or special pass to the users who didn't enjoy the ride, and so forth.

Challenges

Passbook has really matured as a technology in iOS 7; Apple has clearly shown that they are willing to keep developing Passbook as a core part of iOS.

The Black Friday pass is much more polished and more interactive. Using the app, especially in connection with the pass, is a much more interesting and connective experience.

Well, it's not over until it's over; if you don't want to end your Passbook experience just yet, try this little follow-up challenge on your own.

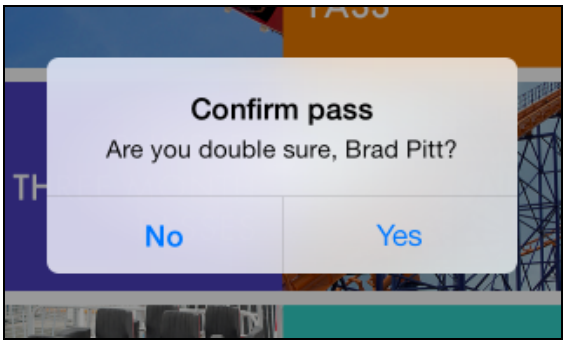
Challenge 1: Reading the user info programmatically

Imagine that during the purchase process of the Black Friday pass your server also includes the customer name within the pass data.

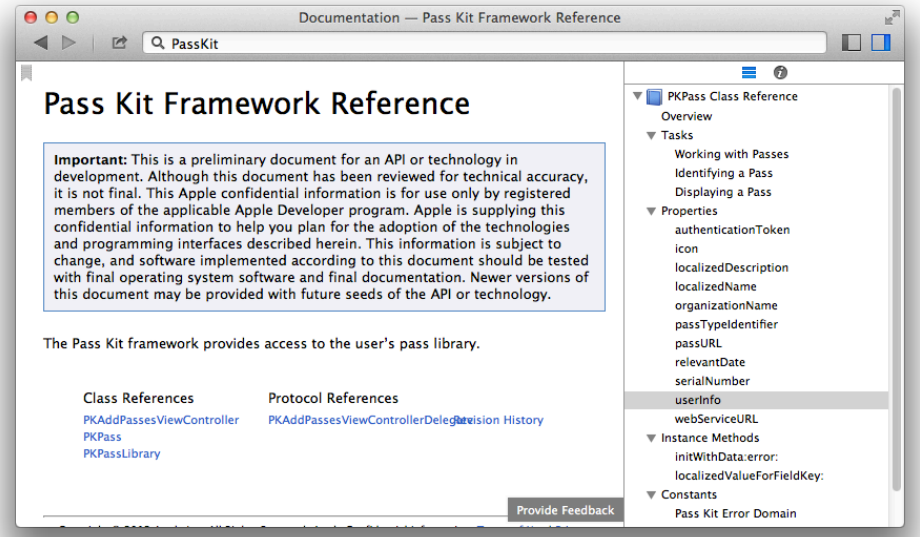
Do this yourself manually:

- Add a new top-level key in the JSON called **userInfo** – make it a dictionary.
- Within userInfo add a key called `customerName` and set a string value for it containing the imaginary customer's name for example "Brad Pitt".

Your task is to modify the Crazy Rides app so that it asks the user whether they are sure they want to import the Black Friday pass into their library and call the user by their name. Keep the message short and funny, like "Are you double sure, <Customer Name>?", like so:



To find the class reference for `PKPass` from Xcode's menu choose **Help/Documentation and API reference**, and search for *PKPass*:



To add the required feature you'll have to modify **MasterViewController.m**:

- Remove the current code from `actionBuy3MonthPass:`
- Fetch an instance of the Black Friday pass – use `passWithName:` to get a `PKPass` object.
- Using the pass object get the **customerName** key out of the `userInfo` property of `PKPass`.
- Now show a `UIAlertView` to ask the user if they want to import the pass, show two buttons "Yes" and "No" as on the screenshot above. Set the delegate to `self`, so you can handle a tap on **Yes**.
- Finally add an `alertView:didDismissWithButtonIndex:` method, and if the tapped `buttonIndex` was 1 invoke the following:

```
[self showPassWithName:@"bf3monthspass"]
```

Good luck with this challenge and your future Passbook adventures!

Chapter 29: Introduction to iAd

By Cesare Rocchi

If you want to monetize your app, there are three primary models to choose from:

1. **Make Your App Paid**. This is the most straightforward route; simply assign a price to your app and be done with it. However, users find the "free" price point to be most attractive.
2. **Add In-App Purchases**. Another great option is adding in-app purchases – whether your app is paid or "freemium". However, this can require a lot of work, as you have to add a bunch of code, set up an in-app store, handle receipt validation, and design your app with in-app purchases in mind.
3. **Add Advertisements**. This approach combines the best of both worlds: you can launch a free app, ask users for their time instead of their money, and still create a passive revenue stream.

Apple has a built-in ad network called iAd that has been around since iOS4. iAd is an easy way to display ads sold by Apple through your app. These ads tend to be high quality and generous revenue share (70% to app developers), but sometimes have a low fill rate.

When the user taps an iAd, it expands to full screen. When the iAd is closed, the user returns to where they left off in the app. This results in a very streamlined user experience — probably the best advertisement model in the industry.

With a proven ad distribution and display model and a generous revenue-sharing program, iAd is arguably one of the most valuable tools in your monetization strategy.

In this chapter, you will integrate iAd into a sample project called "Pushitup", an app to help you track your progress on push-up workouts. Along the way, you'll learn about the various types of iAds, how to sign up for the iAd network, and how to enable iAd in your applications.

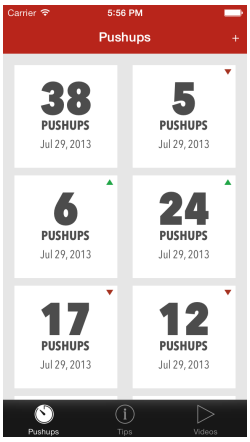
Getting Started

In the resources for this chapter, you will find a starter project for the app with the user interface pre-created – but without any iAds.

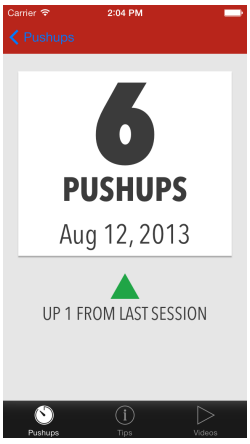
The main view controller contains a tab bar with three sections: Push-ups, Tips and Videos.

- **Push-ups** displays the results of previous workout sessions in a collection view. Tapping a single session displays the session details in a single view. Tapping the **+** button creates a new session to store the number of push-ups and the date of the session.
- **Tips** displays a list of tips to help you improve your push-up technique. Tapping an entry in the list displays the title, date, and full text of the tip.
- **Videos** displays a list of videos available in the app which explain the finer points of push-up technique.

Build and run the project; play around with it for a bit to get a feel for how the app works. Doing the push-ups in real life is optional – but it couldn't hurt! :] This is the first view controller, showing the list of push-up sessions.



When you tap a push-up you can see its details.

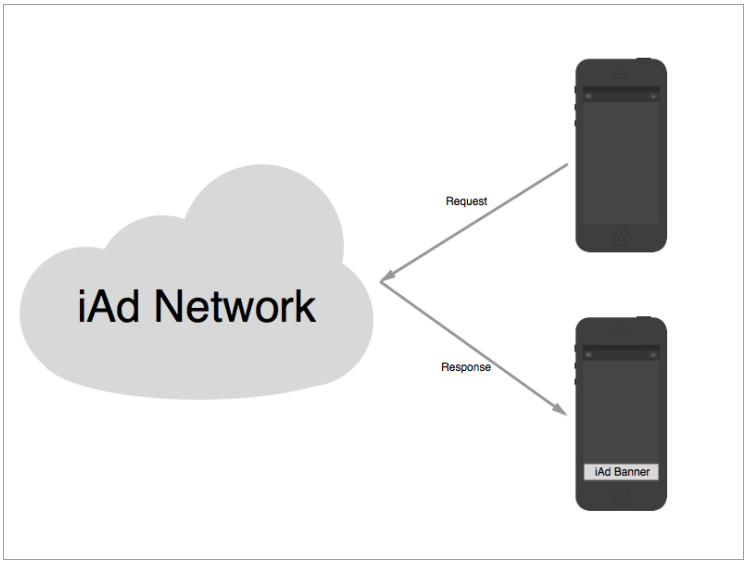


The other sections are organized in a similar fashion: a view controller with a list of items and a detail view controller to show the specifics of an item.

Before you dive into code, first let's talk a little more about iAd and what options you have when working with it.

The iAd workflow

At its core, iAd is an advertising network which many companies use to publicize their products and services. Here's the workflow for iAd behind the scenes:

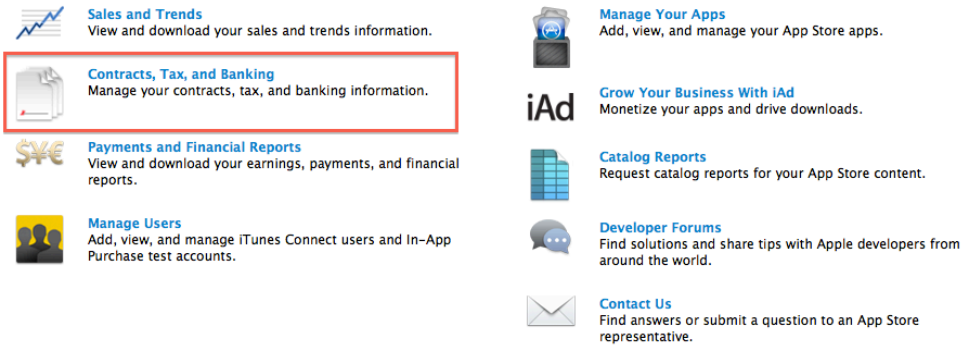


1. A company signs a contract with Apple to publicize their product or service.
2. You sign a contract with Apple to show ads within your application.
3. You enable your application to show iAds.
4. The iAd framework delivers advertisements to fill the advertising spaces you have placed in your application.
5. You earn money when your customers view or interact with the advertisements.

If you're like most developers, you like to get the paperwork out of the way as quickly as possible so that you can get started coding. The next section quickly walks you through the signup process so that you can get busy extending your app to support the iAd framework!

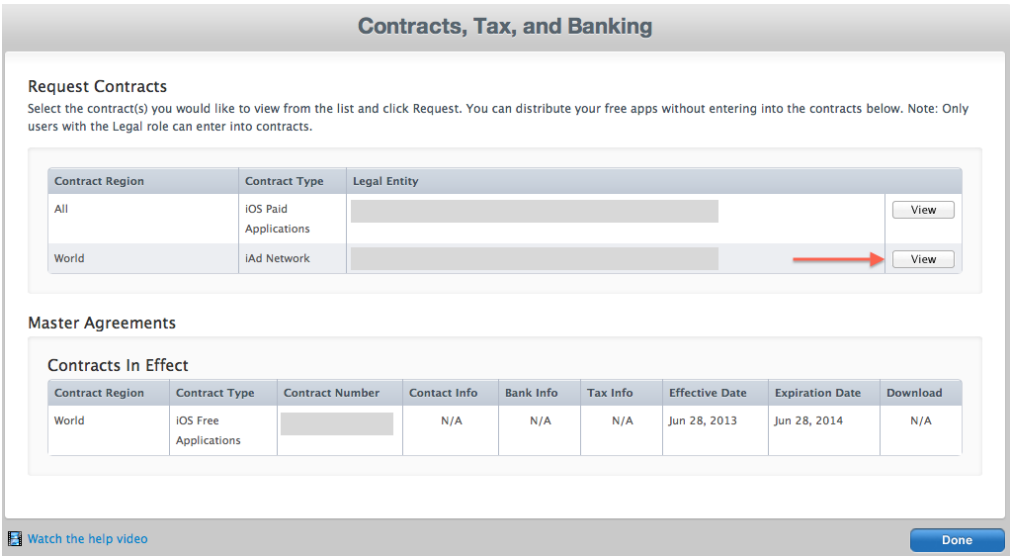
Signing up for iAd

Log in into http://itunesconnect.apple.com as the administrator of your account and click on **Contracts, Tax, and Banking**, as indicated below:



Check that the "Contracts in Effect" section includes an entry for "iAd Network" as shown in the following screenshot:

If you don't see an entry for iAd, then simply apply for the iAd contract by clicking the view button corresponding to the iAd contract and follow instructions.



You only have to perform this step once; occasionally Apple changes clauses in the iAd developer agreement and prompts you to accept these changes on your iTunes Connect page, but beyond that there's no further action required to sign up for iAd.

That's it for the necessary paperwork; you're now free to start implementing iAd in your apps!

Integrating iAd into your app

It's quite easy to integrate iAd into your app. You simply provide "spaces" in your application and link to the iAd framework. At runtime, the iAd framework queries the network for an advertisement with your preferred content and sized to fit the ad formats in your application.

Each view in your application is a potential home for an advertisement. Although sticking an ad on every view sounds like the best revenue-generating strategy, it's a poor idea in practice. The user experience of your app is paramount, and you don't want to compromise the usage of your app just to shoehorn in a few ads.

Instead, you'll want to use high-traffic and high-visibility areas of your app, such as views that are used frequently when your app is running, or "loading" screens before content. Similarly, you want to avoid areas in your app that aren't frequently used. For example, your app's "Settings" view isn't opened very often, so placing an ad there is likely a waste of effort.

It's also important to not overuse ads (such as placing more than one ad in a single view), or to try to trick users into tapping ads. Your user experience will suffer as a result – and your app is likely to be rejected from the App Store.

Types of Advertisements

There are four types of advertisements available in the iAd framework:

1. Banner
2. Interstitial
3. IAB medium rectangle
4. Pre-roll video

The **banner** is the classic in-app advertisement. It displays a narrow strip at the top or bottom of your view that extends the full width of the screen. This strip has different heights depending on the device:

- **iPhone and iPod Touch, portrait**: 50 points
- **iPhone and iPod Touch, landscape**: 32 points
- **iPad, all orientations**: 66 points

When the user taps the banner, it shows a full-screen interactive advertisement. When the user closes the ad, they automatically return to their original place in the app.

The **interstitial** is a full-screen advertisement shown between view transitions. For example, when a user selects an item in a table view, the app shows an interactive full-screen advertisement as soon as the new view controller is pushed. When the user closes the ad, the table's detail view controller is revealed.

The **IAB medium rectangle** works much like a banner with fixed dimensions of 300 by 250 points. This type of ad works well when placed inline with other components, such as in the middle of a scroll view.

> **Note**: Wondering what IAB stands for? This refers to the Interactive Advertising Bureau which maintains a list of standard ad sizes. The 300x250 ad size is known as a "medium rectangle" or "big box" in industry-speak.

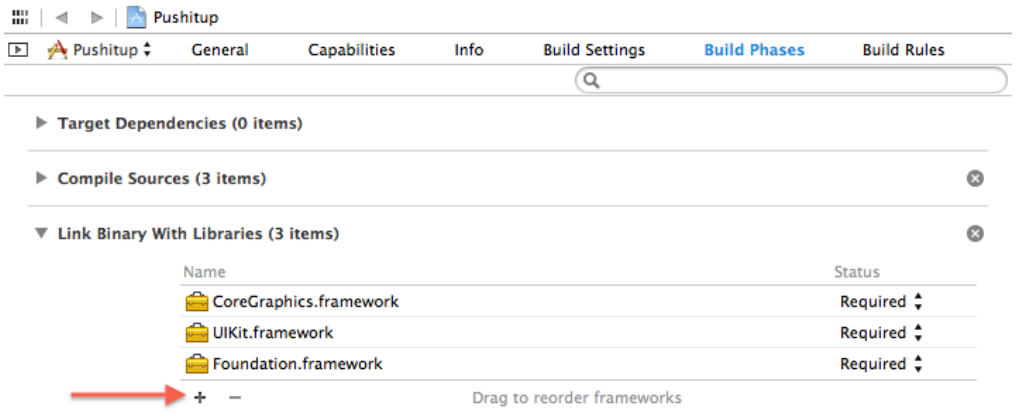
The **pre-roll video** is brand-new in iOS 7. It's very similar to the video advertisements on YouTube that play before your requested video is streamed. After a few seconds the pre-roll gives the user the option to engage with the video or skip it and move on to the main content.

The best way to learn about each of these ad types is by getting your hands dirty and implementing each of them in your sample project. The following section shows you how to link the iAd framework to the starter app and get started in the world of in-app advertising!

Linking to the iAd framework

Before you do anything with iAd, you first need to add the iAd framework to your project and import its header file.

To do this, select the root of the project, click the **Build Phases** tab and expand the section **Link Binary with Libraries**. Tap the **+** button and select **iAd.framework** from the list.



Since ads will be displayed in different areas of the app, it makes sense to import the framework into the pre-compiled header. Open **Supporting Files\Pushitup-Prefix.pch** and add the following import statement under the other `#import` statements:

```
@import iAd;
```

That's it – you're ready to work with iAd!

Adding a banner

The banner is the simplest form of advertising in the iAd platform and is available on iPhone, iPod Touch and iPad targets.

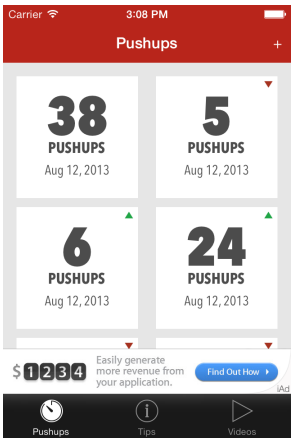
Previous versions of iAd required some fiddly code to rearrange the views of an app and make room for the banner[1]. In iOS 7, adding a banner to a view controller is now extremely simple.

Next, you have to tell iOS 7 to display an ad in the view controller of interest. To do this, open **PushupListViewController.m**, and add the following line of code to the end of `viewDidLoad`:

```
self.canDisplayBannerAds = YES;
```

That's all you need! The above code displays a banner ad at the bottom of the view controller; that's Apple's recommended location for banner ads. `canDisplayBannerAds` instructs iAd to enable ads on this view controller and resize the view as necessary.

Build and run your application; you should see a banner ad at the bottom of the screen, as demonstrated in the following screenshot:



[1] Take a look at this tutorial for an example: http://www.raywenderlich.com/1371/how-to-integrate-iad-into-your-iphone-app

Scroll to the bottom of the collection view; you'll notice that the banner is not simply an overlay that covers a portion of the view. The original view has been automatically resized to make room for the banner.

Here's what's happening under the hood:

- Setting `canDisplayBannerAds = YES` wraps the original view in a property called `originalContentView`.
- Once loaded, the framework moves the banner onto the screen and resizes `originalContentView` accordingly.
- When the banner disappears, the framework resizes the view to its original dimensions.

Tapping on the banner displays the full-screen ad and calls `viewWillDisappear:`. This method is the perfect place to halt any intense action in your app or pause the playback of audio or video files that might clash with the ad media.

Dismissing the full screen ad calls `viewDidAppear:` where you can resume any paused activities or media playback.

> **Note:** The loading and refreshing of ads is managed completely by the iAd framework. Your code doesn't need to request new banners; it's completely automated and new banners are displayed as they become available.

Adding interstitial ads

Unlike banners which take up a small amount of space, an interstitial advertisement fills the screen. iOS 4.3 introduced this type of ad to the iPad, and iOS 7 brings interstitial ads to the iPhone.

Interstitial advertisements appear when transitions occur in your app. For example, when the user selects a push-up tip in the list view, the app transitions from `TipListViewController` to `TipViewController`. iAd listens for this transition and displays an advertisement just as `TipViewController` becomes visible.

Open **AppDelegate.m** and add the following line to `application:didFinishLaunchingWithOptions:` just before the `return` statement:

```
[UIViewController prepareInterstitialAds];
```

This call pre-fetches interstitials in the background to have them ready for transitions. Since view transitions are fairly quick, it's critical to call `prepareInterstitialAds` as early as possible in your application's lifecycle to give the framework enough time to retrieve and prepare the ads for display.

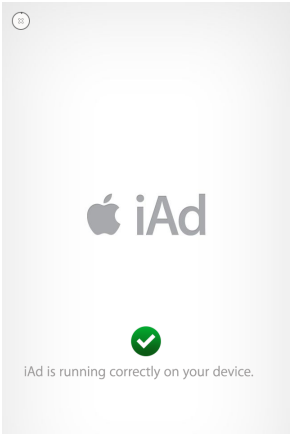
Now you need to give the iAd framework some information about the transitions to listen for.

Open **TipListViewController.m** and modify `prepareForSegue:sender:` as shown below:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue
                 sender:(id)sender
{
    if ([[segue identifier] isEqualToString:@"showDetail"]) {
        NSIndexPath *indexPath =
                    [self.tableView indexPathForSelectedRow];
        Tip *tip = self.tips[indexPath.row];

        // add these lines below
        UIViewController *c = segue.destinationViewController;

        if ([[UIDevice currentDevice] userInterfaceIdiom] ==
            UIUserInterfaceIdiomPhone) {
            c.interstitialPresentationPolicy =
                      ADInterstitialPresentationPolicyAutomatic;
        }
        // end lines to add
        [[segue destinationViewController] setTip:tip];
    }
}
```

In the above code, you check the device type to ensure that interstitials will only display on an iPhone. Then you set `interstitialPresentationPolicy` to `ADInterstitialPresentationPolicyAutomatic` which tells the framework that a transition from `TipListViewController` to `TipViewController` is a candidate for an interstitial advertisement.

Build and run your app on an iPhone or iPhone simulator; switch to the **Tips** tab, select one of the list items and check out the full-screen interstitial ad.



Return to the list view and select a few more items in the list. You'll notice that the interstitial ad doesn't appear for every transition to `TipViewController`. What gives?

Ads are displayed according to the policy set at design time. `ADInterstitialPresentationPolicyAutomatic` gives the framework full control of the timing of ads; to manage the timing yourself, set the policy to `ADInterstitialPresentationPolicyManual`. When the policy is manual you have to explicitly request an ad using the method `requestInterstitialAdPresentation`, as in the following code snippet.

```
UIViewController *c = segue.destinationViewController;
c.interstitialPresentationPolicy =
              ADInterstitialPresentationPolicyManual;
[c requestInterstitialAdPresentation];
```

This will allow you to display a banner when you think it's appropriate.

Adding IAB medium rectangle

The IAB medium rectangle banner is quite similar to the regular banner ad; however, this ad type is only available on the iPad. It has a predictable size of 300 by 250 points, cycles through different ads, and shows a full screen interstitial ad when tapped.

`TipViewController` is a great candidate for this type of ad; there's room to show an ad directly beneath the date label.

Open **TipViewController.h** and add the following line to the class definition:

```
@interface TipViewController : UIViewController
    <ADBannerViewDelegate>
```

This implements the `ADBannerViewDelegate` protocol in `TipViewController`.

Next, open **TipViewController.m** and add the following code to the end of `viewDidLoad:`

```
    if ([[UIDevice currentDevice] userInterfaceIdiom] ==
       UIUserInterfaceIdiomPad) {
        self.bannerView = [[ADBannerView alloc]
          initWithAdType:ADAdTypeMediumRectangle];
        self.bannerView.delegate = self;
        self.bannerView.center = self.view.center;
        self.bannerView.hidden = YES;
        [self.view addSubview:self.bannerView];
```

```
    }
```

The above code checks to see if the app is running on the iPad. If so, then it sets the ad type, centers the ad horizontally and vertically. Finally, it adds the advertisement view by calling `addSubview:`.

Next, add the following two methods (still in **TipViewController.m**):

```
- (void)bannerViewDidLoadAd:(ADBannerView *)adView {
    self.bannerView.hidden = NO;
    [self updateUI];
}

- (void)bannerView:(ADBannerView *)banner
  didFailToReceiveAdWithError:(NSError *)error {
    NSLog(@"banner failed loading");
    self.bannerView.hidden = YES;
    [self updateUI];
}
```

The first method triggers on a successful ad request and shows the ad by setting the `hidden` property of the advertisement subview to `NO`. The second method triggers on an unsuccessful ad request and hides the subview by setting `hidden` to `YES`. The user is never aware that the banner didn't load successfully. Both methods include a call to `updateUI`. This method resizes the text view to make room for the ad when it's loaded or sets the text to its original dimensions when the ad fails to load.

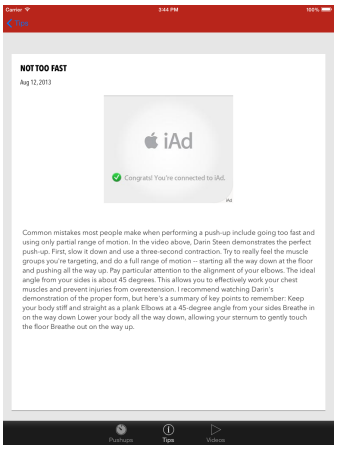
```
- (void) updateUI {

    if (!self.bannerView.hidden) {

        self.bannerView.center = self.view.center;
        CGRect adFrame = self.bannerView.frame;
        adFrame.origin.y = self.dateLabel.frame.origin.y +
                           self.dateLabel.frame.size.height + 70;
        self.bannerView.frame = adFrame;

        CGRect textFrame = self.bodyTipTextView.frame;
        textFrame.origin.y = adFrame.origin.y +
                             adFrame.size.height;
        textFrame.size.height -= adFrame.size.height+ 20;
        self.bodyTipTextView.frame = textFrame;

    } else {
```

```
        CGRect textFrame = self.bodyTipTextView.frame;
        textFrame.origin.y = self.dateLabel.frame.origin.y + 70;
        textFrame.size.height =
                            self.containerView.frame.size.height –
                            textFrame.origin.y -20;
        self.bodyTipTextView.frame = textFrame;
    }
}
```

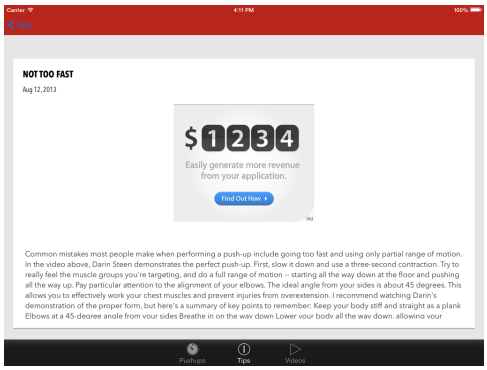
As a final touch you should re-layout the view also when the device rotates. Still in **TipViewController.m** add the following method.

```
-(void)willAnimateRotationToInterfaceOrientation:
    (UIInterfaceOrientation)toInterfaceOrientation
                duration:(NSTimeInterval)duration {
    [self updateUI];
}
```

Build and run your app on an iPad or iPad simulator; select the Tips tab and tap a single tip in the list. You should see your ad displayed in the middle of the tip text, as shown in the following screenshot:



Rotate the device in landscape and you will see the following layout.



Turn on Airplane mode to simulate a failure with the iAd network and repeat the steps above. You should see the log statement appear in the console and the banner view should not be visible, showing no indication that the ad request failed.

Pre-roll video advertisements

If your app includes the functionality to display videos you can show video advertisements before your video content. We can increase the revenue of Pushitup by displaying video ads before playing the video tips included in the app.

Pre-roll advertisements are integrated through the class `MPMoviePlayerController`[2]; it's been included in the starter project for you. As with interstitials, the app needs to perform a pre-fetch step for the pre-roll ads as early as possible

Open **AppDelegate.m** and add the following line to `application:didFinishLaunchingWithOptions:`, right before the `return YES` statement:

```
[MPMoviePlayerController preparePrerollAds];
```

This tells the iAd framework to begin fetching the pre-roll advertisements.

Next, open **VideoListViewController.m**, locate the following line in `tableView:didSelectRowAtIndexPath:`

```
[self.moviePlayerController play];
```

…and replace it with the following code:

```
[self.moviePlayerController
   playPrerollAdWithCompletionHandler:^(NSError *error) {

 if (error)
   NSLog(@"error in playing preroll");

 [self.moviePlayerController play];

}];
```

`playPrerollAdWithCompletionHandler:` plays the pre-roll ad if one is available. When the pre-roll finishes playing or is interrupted by the user, the completion

---

[2] If you are not familiar with this class you can check out the sample code provided by Apple here: http://developer.apple.com/library/ios/#samplecode/MoviePlayer_iPhone/Introduction/Intro.html#//apple_ref/doc/uid/DTS40007798

handler is called and control returns to this block of code. Any errors are logged and video playback begins.

Much like interstitials, pre-rolls are only shown if the video is available. Therefore you might not see them every time you play your main video.

The following screenshot demonstrates what the user will see when a pre-roll plays:

At this point you've covered the full range of ads available, but you might want to tweak the frequency with which ads are rotated to see how different ads appear in your app. The next section describes how to control this and other options for developer-mode apps.

Settings for testing

When you are testing your app embedding iAd you probably prefer to tweak the way ads are loaded from the network. Sometimes you might need a very high frequency, sometimes a very low one. There are a several developer settings on your device that control the fill rate and refresh rate of iAd.

Open the Settings app on the device and select the **Developer** entry shown in the screenshot below:

This lets you modify the parameters below which affect the display of ads:

- **Fill Rate** – sets the percentage of requests for ads
- **Refresh Rate** – sets the frequency with which ads are refreshed
- **Highlight Clipped Banners** – when turned on it highlights clipped banners: green means everything is ok, red means the banner is clipped. This is useful especially for IAB Medium rectangles which are placed in line with content.
- **Unlimited Ad Presentation** – when turned on sets to zero the time between interstitial and pre-roll ads presentation. Turn it on during development so you don't need to wait before presenting another ad when you are testing.

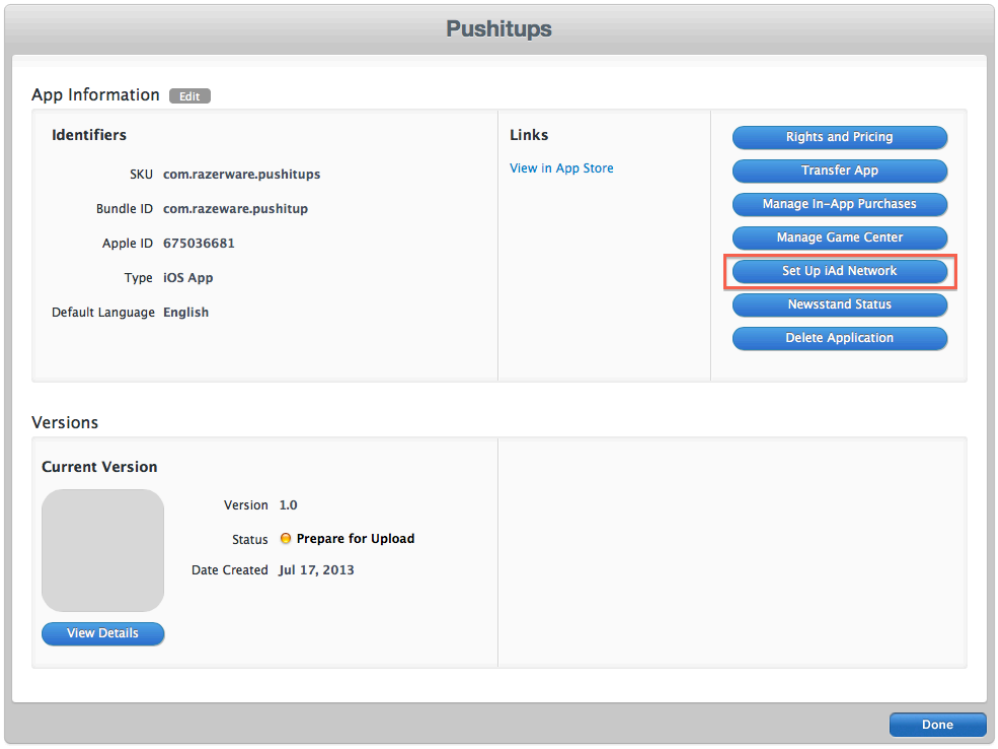
Note that these settings only affect applications running in developer mode; these settings don't influence the behavior of other installed apps.

You've covered all of the critical bits of working with iAd; the only thing left to do is enable the iAd network when publishing your app to the App Store.

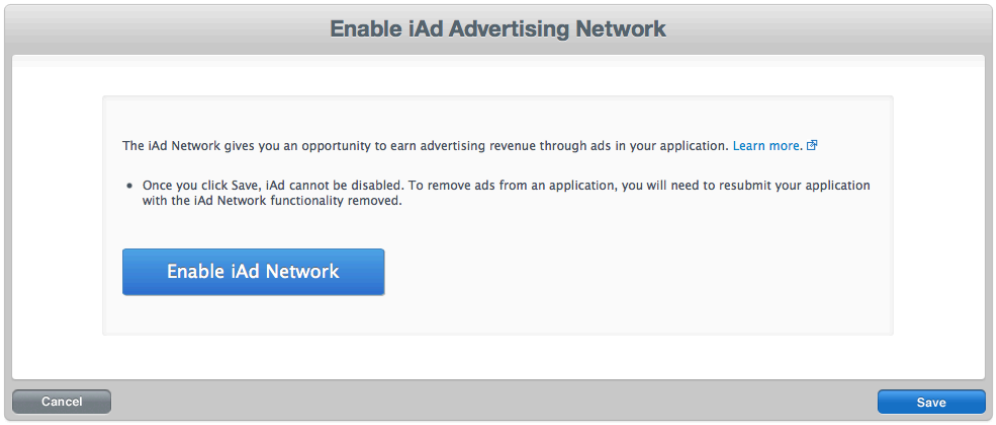
Enabling iAd in your app

Ads won't automatically show up in your app unless you enable iAd in iTunes Connect before publishing to the App Store.

Once you've entered all of the requisite information as part of the submission process, you'll arrive at the "Prepare for Upload" state. Click on the **Set up iAd Network** button on the right as shown below:



You'll be presented with a dialog that invites you to enable iAd for your app. Click the blue **Enable iAd Network** button, then the **Save** button, as illustrated below:



Now iAd is enabled for your app and you can proceed with the rest of the submission process. If you ever need to disable iAd in your app, you'll need to go through the submission process again and ensure that iAd is disabled for the new version of your app.

Adding a banner programmatically

You might not be happy with the default position of banners. For example, you think it would be better to show a banner at the top of a table view. In this case it's not enough to set the `canDisplayBannerAds = YES`. You have to write some code.

In this section you will learn how to add a banner view as a header in a table view. Start by opening **VideoListViewController.m** and add change the declaration as follows:

```
@interface VideoListViewController () <ADBannerViewDelegate>
```

Below the properties already declared add two more.

```
@property (nonatomic, strong) ADBannerView *bannerView;
@property (nonatomic, assign) BOOL adLoaded;
```

At the end of `viewDidLoad` add these two lines to initialize the banner view and set the video view controller as delegate.

```
self.bannerView = [[ADBannerView alloc]
                           initWithAdType:ADAdTypeBanner];
self.bannerView.delegate = self;
```

Next add the following methods of the delegate

```
- (void)bannerViewDidLoadAd:(ADBannerView *)banner {
    NSLog(@"did load");
    self.adLoaded = YES;
    [self.tableView reloadData];
}

- (void)bannerView:(ADBannerView *)banner
  didFailToReceiveAdWithError:(NSError *)error {
    NSLog(@"error in loading banner");
    self.adLoaded = NO;
    [self.tableView reloadData];
}
```

The first marks the banner as visible, while the second hides it. Both contain a call to tableView's `reloadData` to refresh the table when an ad is loaded. Next, set the banner view as the view for the table's header, by adding the following method

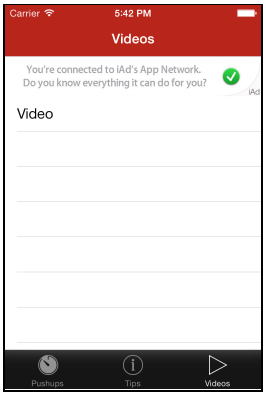
```
- (UIView *)tableView:(UITableView *)tableView
  viewForHeaderInSection:(NSInteger)section {
    return self.bannerView;
```

```
}
```

Finally, implement the tableView:heightForHeaderInSection: to dynamically set height of the header according to the status of the ad (loaded or not) and the type of device[3].

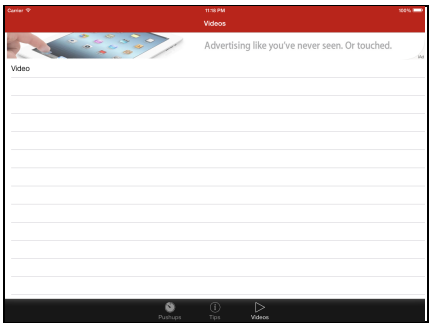
```
- (CGFloat)tableView:(UITableView *)tableView
  heightForHeaderInSection:(NSInteger)section {
  if (self.adLoaded) {
    if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) {
      return 66;
    } else {
      return 50;
    }
  }
  return 0;
}
```

Build and run the application and the video list view controller will show a banner as the header of the table view.



Test it also on the iPad and notice that the resize of the width is automatically managed.

[3] The size of banners changes according to the device. Check this link for more information: http://developer.apple.com/library/ios/documentation/userexperience/conceptual/iAd_Guide/BannerAdvertisements/BannerAdvertisements.html#//apple_ref/doc/uid/TP40009881-CH3-SW8



Adopting the same technique you can place banner ads wherever you like in your application. The skeleton to adopt is the following

1. Initialize an instance of `AdViewBanner`
2. Implement the delegate methods `bannerViewDidLoadAd` and `bannerView:didFailToReceiveAdWithError:`.
3. Update the UI taking into account the state of the banner, visible or hidden.

Now it's time to move to another advanced topic: ad mediation.

Ad Mediation

iAd is a great resource for monetizing an iOS application. Advertisements are carefully curated and do not break the user experience of the app.

However, iAd isn't perfect – sometimes it does not have enough fill rate to display ads for all the impressions you might have. You might want to display iAds when they're available, and ads from other networks the rest of the time.

To do this, you can use an ad mediation framework, which allows you to display ads from different networks (including iAd) within your app easily. An ad mediation framework typically consists of two parts:

- A client SDK that you integrate into your app
- A server-side component that serves ads to your app

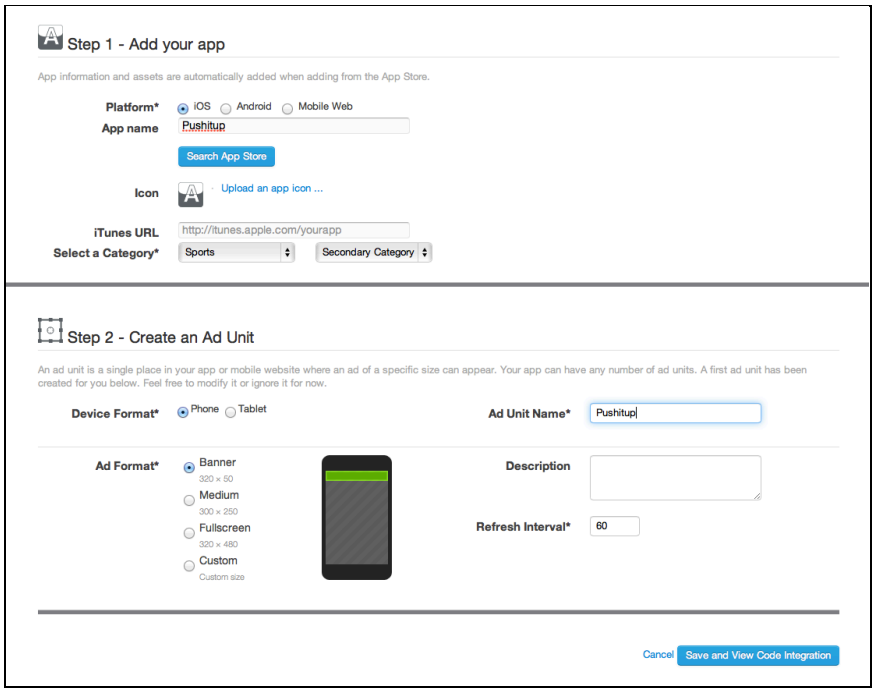
To use an ad mediation framework, you typically create an account, set up an application on their web site, and install the SDK in your app. Then you can activate different ad networks and tweak the ratio of advertisements between the networks. For example, you could set up the platform to serve iAd 70% of the time and ads from another network for the remaining 30%.

In this section you will learn about one popular ad mediation framework (mopub.com) and integrate it Pushitup. You will configure ads to be served by two different networks, iAd and Millennial Media (http://www.millennialmedia.com).

Creating a Mopub account and app

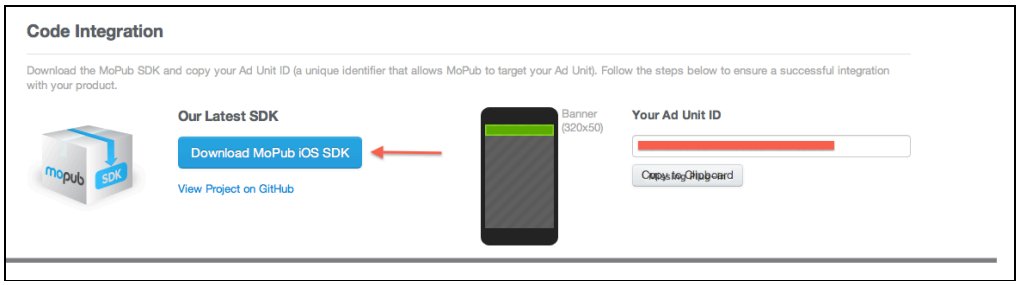
The first step is to create an account on Mopub. Visit http://www.mopub.com, click on the Sign Up button on the top right and follow the instructions.

Once you are done visit this link https://app.mopub.com/dashboard/ and login with your credentials. Now click on the "Inventory" link at the top and you will end up on this screen.



Enter the details of your app, the category, you choose **Phone** as device format and **Banner** as ad format. Then click **Save** at the bottom right.

After you finish creating your app, you should see your **Ad Unit ID** listed on the page. Copy it and save it somewhere because you will need it later.



You'll also see a big blue button that says **Download MoPub iOS SDK**. Let's do that next!

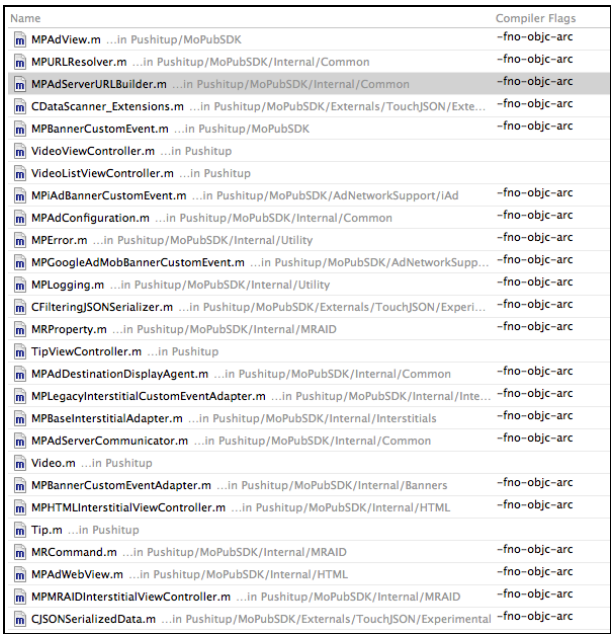
Integrating the SDK

Click the blue button on the left to download the SDK.

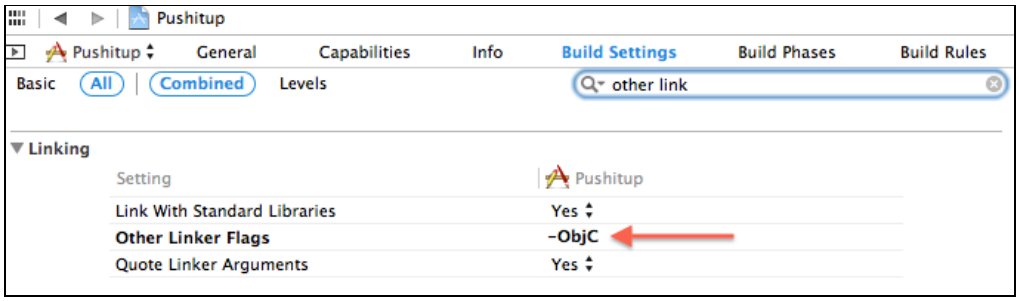
Unzip the file and drag the whole folder on the root of the Xcode project. Then select **Project Root / Target / Build Phases** and link against the following frameworks:

- QuartzCore
- MobileCoreServices.framework
- MessageUI.framework
- MediaPlayer.framework
- CoreTelephony.framework
- CoreLocation.framework
- AVFoundation.framework
- AudioToolbox.framework
- AdSupport.framework

Still in Build Phases expand **Compile Sources**, select all the files in the folder **MoPubSDK** and add the compiler flag **-fno-objc-arc** as in the following screenshot:



Select the **Build Settings** tab, search for **Other Linker Flags** and add the flag **-ObjC**.



Open **PushupViewController.m**, import MPAdView and modify the interface declaration like this:

```
@interface PushupViewController () <MPAdViewDelegate>

@property (nonatomic, strong) MPAdView *adView;

- (void) configureView;

@end
```

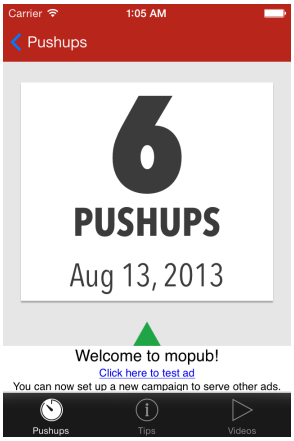
At the end of `viewDidLoad` add the following snippet, using the ad ID saved previously.

```
if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPhone) {
  self.adView = [[MPAdView alloc] initWithAdUnitId:@"YOUR_ID"
    size:MOPUB_BANNER_SIZE];
  self.adView.delegate = self;
  self.adView.center = self.view.center;
  self.adView.hidden = YES;
  [self.adView loadAd];
  [self.view addSubview:self.adView];
}
```

Finally add the following two methods:

```
-(UIViewController *)viewControllerForPresentingModalView {
    return self;
}

-(void)adViewDidLoadAd:(MPAdView *)view {
    CGRect frame = self.adView.frame;
    CGSize size = [self.adView adContentViewSize];
    frame.origin.y = [[UIScreen mainScreen]
      applicationFrame].size.height - size.height - 95;
    self.adView.frame = frame;
    self.adView.hidden = NO;
}
```

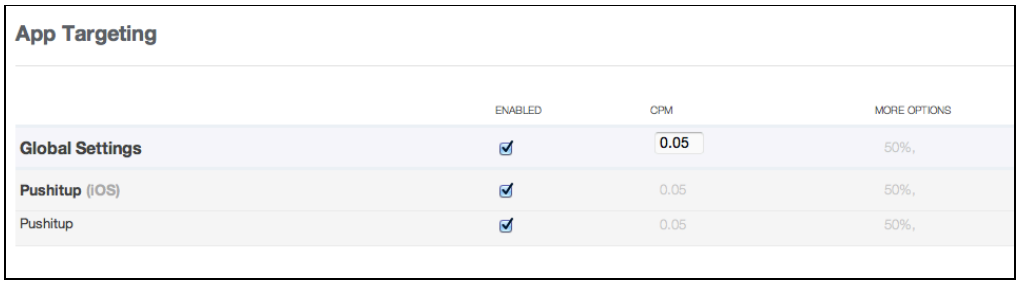
The first sets the current instance of `PushupViewController` as the host for the ad and the second arranges the UI to show the banner. Build and run the application on the iPhone, select a push-up and you will see the test ad from Mopub.



This means you are on the right path. Let's move on the back-end to tweak some settings.

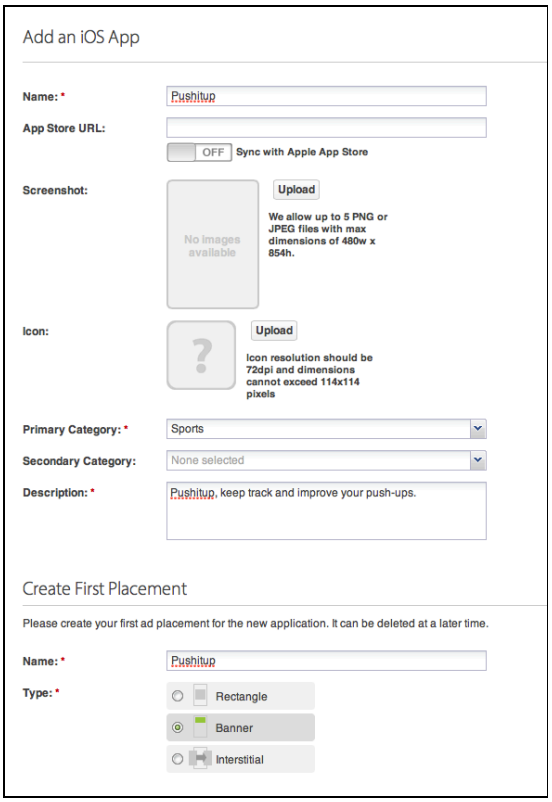
Setting up ad networks

The great feature of an ad mediation network is that you can tweak the appearance of ads by setting properties on a server, without the need to recompile the application. Visit this URL https://app.mopub.com/networks/ and click on the **Set up iAd** button. Scroll to the **App Targeting** section, enable it for the Pushitup application, click on the **More Options** on the right and set **50** for the allocation value. You will end up with this screen:



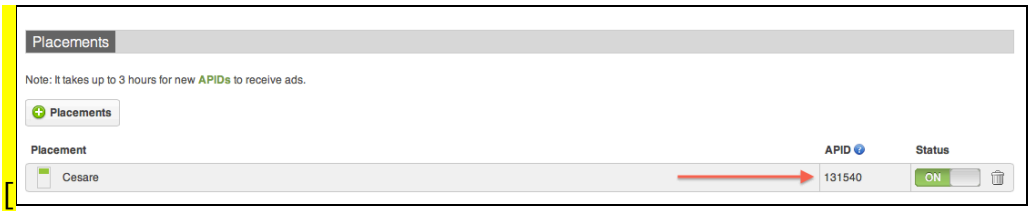
This says that iAd is enabled and the 50% of ads served will be from Apple. Click save at the bottom right. Build and run the application on iPhone or iPhone simulator and repeat the previous test. You will see test ads from both iAd and mopub.

Now let's add the Millennial ad network. If you don't have it already, create an account: visit this URL https://tools.mmedia.com/login/register and follow the instructions. Once you are done login and visit this URL https://tools.mmedia.com/apps/manageApps and tap **Add an App**. Choose iOS as platform and click next at the bottom right. Enter the following details and click **Finish**.

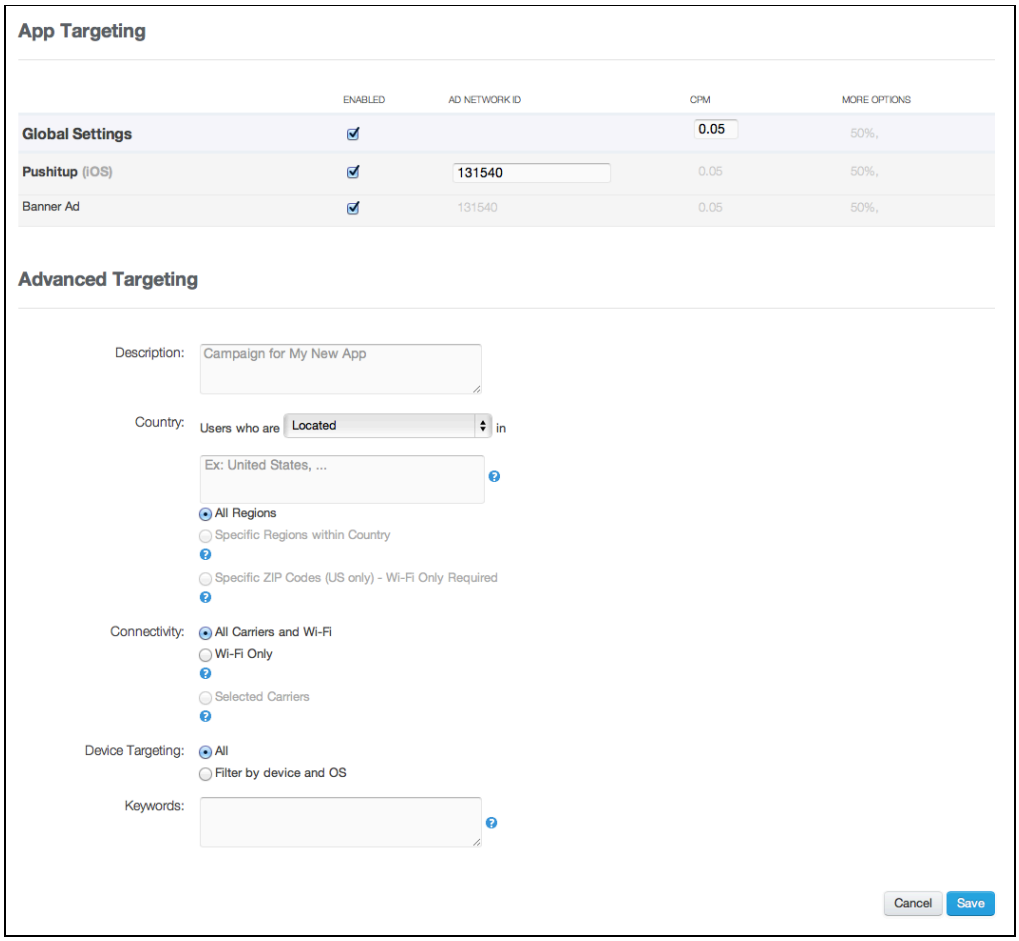


The next step would be the configuration of the SDK but the Mopub SDK downloaded previously already includes the support to the Millenial network. All you need is the API ID of the application you have just created.

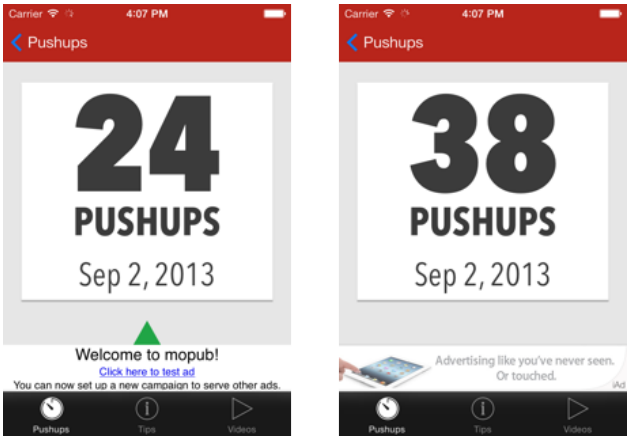
Skip the SDK configuration step, visit https://tools.mmedia.com/apps/manageApps, select the only app in the list, scroll to the bottom and on the right you'll find the **API ID**. Copy it and save it somewhere. You will need it in a few minutes.



Now head back to the Mopub back-end and visit https://app.mopub.com/networks/. On the right choose **Add a Network** and select **Millennial Media**. In the **App Targeting** section enable this network, paste the **API ID** copied previously and set the allocation to 50%. Finally click **Save** at the bottom right.



Build and run the application to check that ads from both iAd and Millennial appear in the push-up detail view:



Note: Millennial takes up to 48 hours to review your app and put it live. During this time you will see test ads.

Congratulations - now you have two networks serving ads to your app! Another cool benefit is you can tweak its advertising settings without the need to release a new version.

Challenges

Advertisements can be a great source of revenue for your apps. At this point, you have a great foundation on how to ad ads to your app, whether via iAds or using an ad mediation framework like mopub.com.

You have learned how to setup an iAd banner view, how to use interstitial ads (automatic and manual), how to show a IAB Medium banner in an iPad application and how to play pre-roll ads before video contents.

You have also covered also tips and tricks, like showing a banner in a table view's header and dynamically re-arranging the layout of a view when a banner is loaded or unloaded. Finally you have learned how to set up and tweak Mopub, an ad mediation platform.

But don't go yet – we have three challenges for you to get some extra experience with iAd!

Challenge 1

Most of the techniques illustrated in this chapter have been implemented via code but you can use Storyboards as well. Here is a challenge for you. In the final project open **PushupViewController.m** and modify `viewDidLoad` as follows:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    self.containerView.layer.shadowColor =
      [UIColor lightGrayColor].CGColor;
    self.containerView.layer.shadowOffset = CGSizeMake(0, 2);
    self.containerView.layer.shadowRadius = 0.5f;
    self.containerView.layer.shadowOpacity = 1.0f;
    [self configureView];

    //self.iAdView.hidden = YES;
    self.iAdView.delegate = self;
}
```

Now when the user opens the view no ad is displayed. Try adding a simple banner at the bottom of the iPhone app using Storyboards. Here are a few hints:

- The banner should appear right above the tab bar, as we have seen in previous examples. Hint: use Auto Layout to make the banner stick to the bottom bar.

- Remember that when the view gets opened there might not be an ad available right away. Hint: hide the banner before rendering the view to avoid displaying a white strip (banner placeholder).
- Remember that the loading of a banner can fail (e.g. when there is no network connection. Hint: set `PushupViewController` as delegate for `ADBannerView` and hide the ad view when the loading of an ad fails.
- Finally, the opposite of the previous case, display a banner when it's loaded successfully.

Feel free to take a peak at the project named "Pushitup-final-Challenge1" to check the solution for this challenge.

Challenge 2

In the example using MoPub, the banner overlays the view. Your challenge is to reposition the elements of the view so that the banner does not overlay any of them. Here are a few hints:

- Much like `AdBannerView`, `MPAdView` has a few delegate methods (in particular `adViewDidLoadAd:` and `adViewDidFailToLoadAd:` that you should implement to know the status of the ad view.
- Use them to make room for the banner by repositioning the other elements.
- A bonus if you use an animation to do it.

Feel free to check out the solution in the project "Pushitup-final-Challenge2". In this case I did not use Auto Layout.

Challenge 3

While working on the second challenge you might have noticed that you do not need to reposition elements on the iPhone 5, because there is enough room to show the banner. Review the solution to challenge 2 and apply animations only when the device is an iPhone 4. To detect the size of the device you can use this handy method:

```
[[UIScreen mainScreen] applicationFrame].size.height
```

You can check out my solution in the project "Pushitup-final-Challenge3".

Now go forth and add some iAds to your apps – and hopefully make some money! ☺