5. Airdrop and Multipeer Connectivity

IN THIS CHAPTER

- Using AirDrop to send files between iOS devices
- How to adapt our own app to take advantage of AirDrop
- Basics of Multipeer Connectivity
- Advertising and dicovering peers
- Sending data and resources among peers
- Authentication and encryption

AirDrop

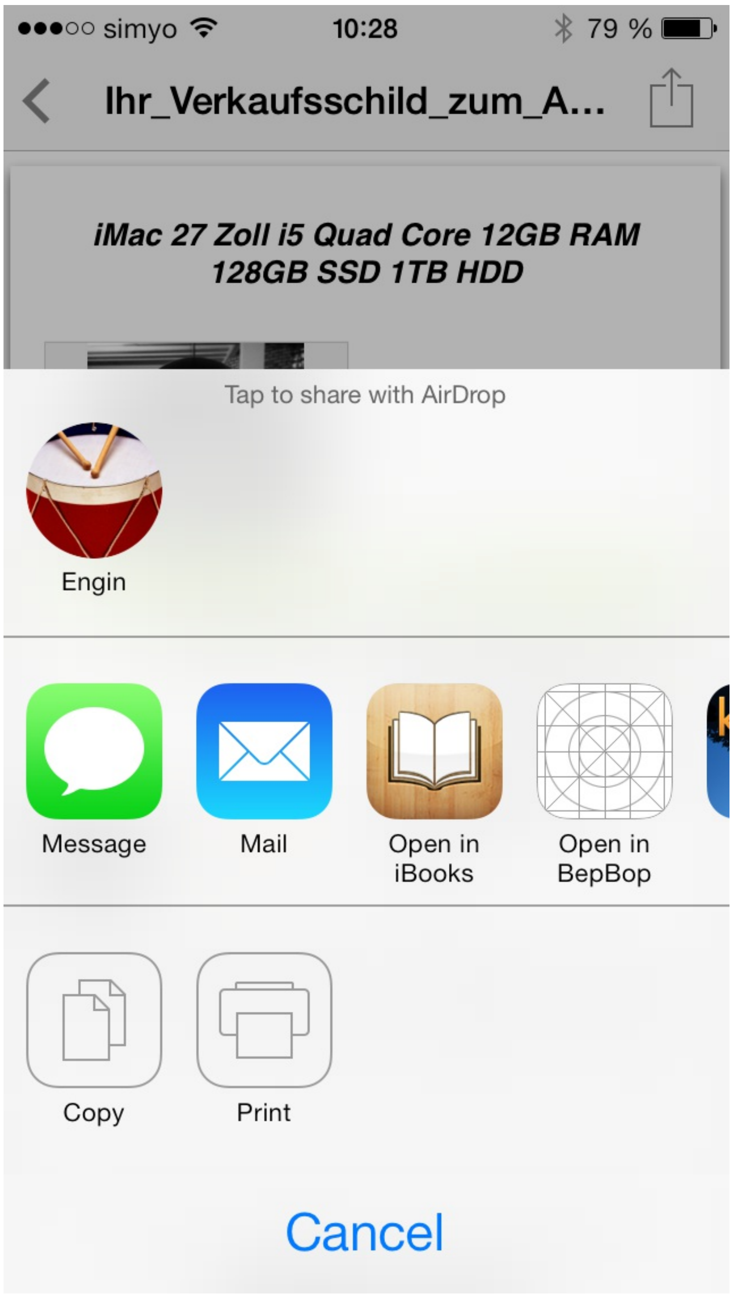
AirDrop is a technology first introduced by Apple in OS X 10.7 Lion. It enables Macs to easily share files. Sharing files between AirDrop capable machines is as easy as activating AirDrop in Finder and dragging the files onto the icon for the desired machine after the nearby computers are shown. AirDrop sharing magically works even if the machines don't have an Internet connection. It even works if the machines are not connected to the same WiFi access point, or to any access point for that matter.

AirDrop works by establishing an adhoc Wi-Fi connection between the participating Macs, thus removing the need for the Internet and an infrastructure wireless network.

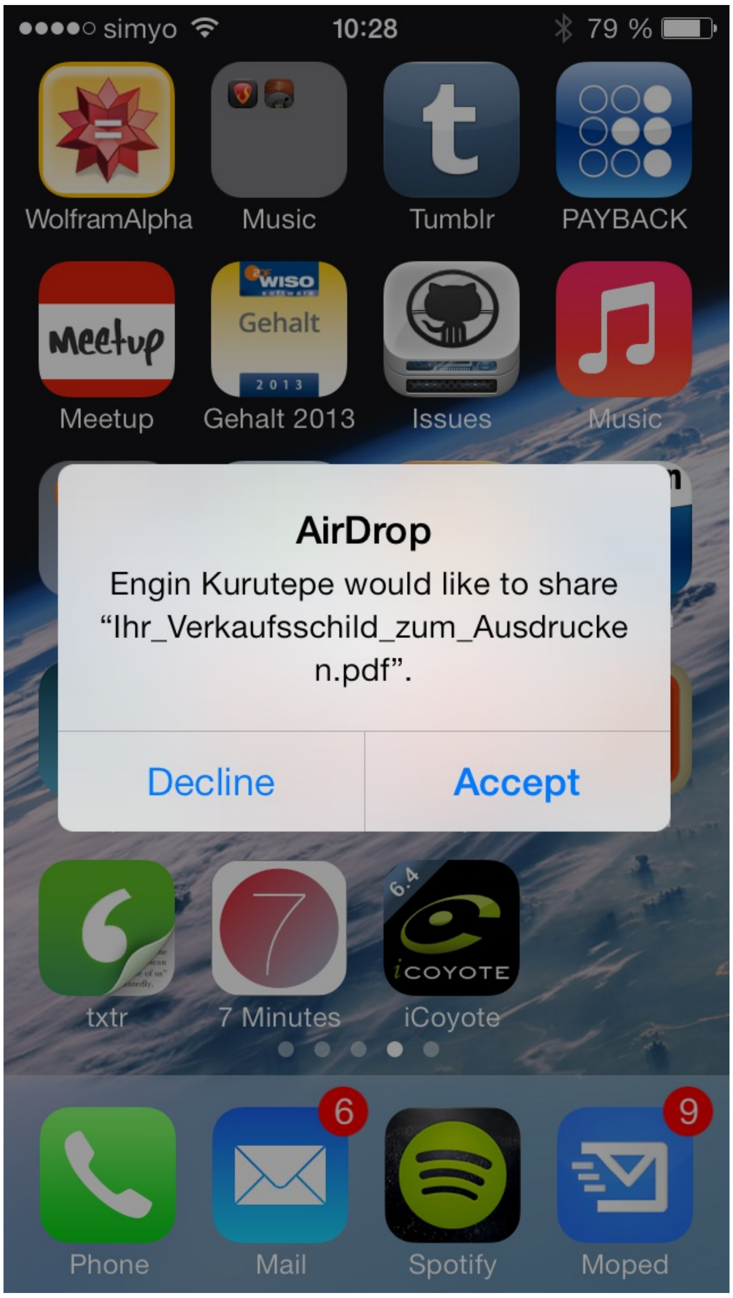
Now, with iOS 7, Apple is bringing this technology to the world of iPhones and iPads. iDevices can share files and documents among each other as long as Wi-Fi and Bluetooth are both turned on (as of this writing, AirDrop on iOS and Mac is not compatible).

How to Use AirDrop

The BepBop app can send and receive PDF files over AirDrop. When you tap to open one of the PDF files listed in the table view in the "Direct Wireless Connectivity" section of the app, you will see an 'action' button at the top right corner of the screen. Tapping that button allows you to send the PDF document over various methods including AirDrop:


●●●○○ simyo
10:28
79 %
Ihr_Verkaufsschild_zum_A...
iMac 27 Zoll i5 Quad Core 12GB RAM 128GB SSD 1TB HDD
Tap to share with AirDrop
Engin
Message
Mail
Open in iBooks
Open in BepBop
Copy
Print
Cancel

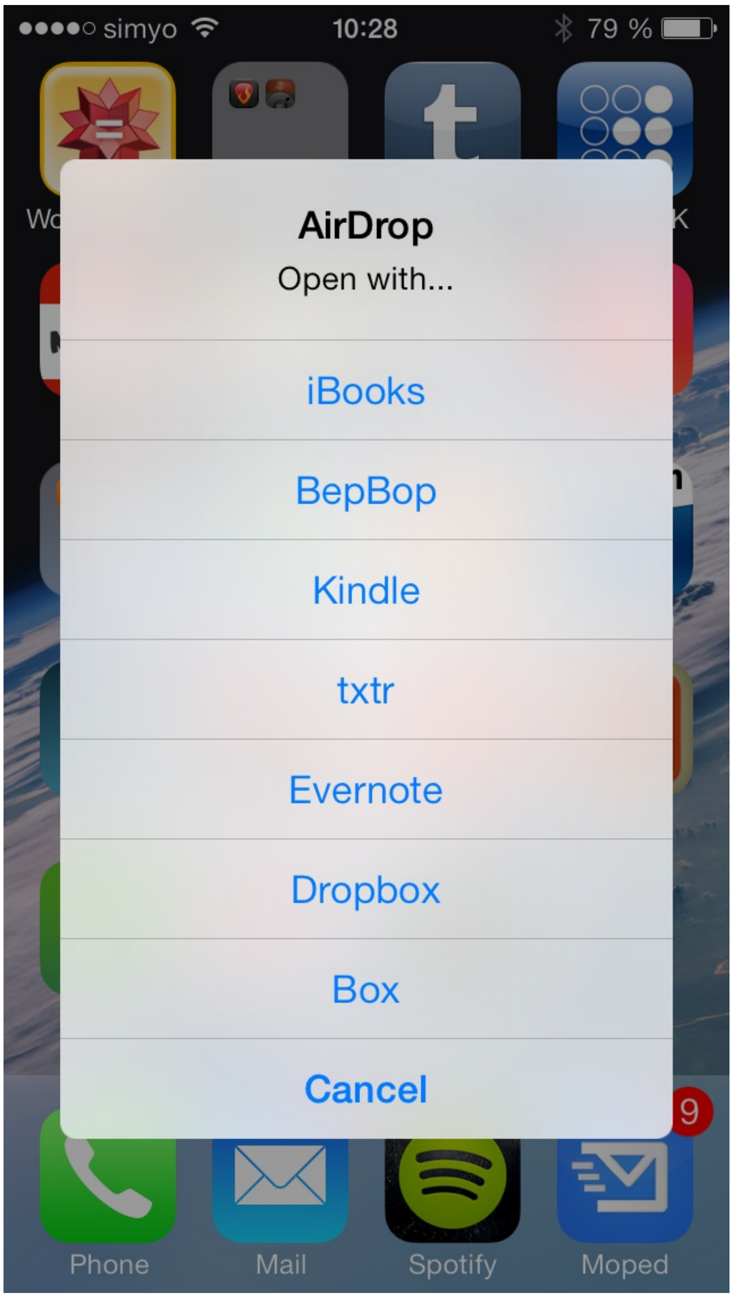
Just tapping on their AirDrop icon sends the file to them and the receiver sees the following confirmation:



AirDrop

Engin Kurutepe would like to share "Ihr_Verkaufsschild_zum_Ausdrucken.pdf".

If there are multiple apps installed that support the same file, they also get to choose which app should open the file you just sent. In this case since we are sending a very common file format, PDF, there are many apps that can handle it, including our demo app, BepBop.



So using AirDrop to share files could not be any easier from a user's perspective. But how do we add AirDrop support to our own apps? In order to show you how this is achieved we build AirDrop support in our demo app, BepBop, and will now walk you through the process.

Adding AirDrop Support to your App

To enable your own app to have the option to share files over AirDrop, you need use the system provided class UIActivityViewController. In the BepBop app we are initilizing an instance of UIActivityViewController in the BEPSimpleWebViewController class:

```
- (void)presentActivities:(id)sender { UIActivityViewController * activities = [[UIActivityViewController alloc]
initWithActivityItems:@[_url] applicationActivities:nil]; [self presentViewController:activities animated:YES completion:nil]; }
```

Where _url is the file URL of the opened PDF file. After initizalization, the activities controller is presented modally. If there are any other devices with AirDrop turned on nearby, the activity controller will automatically show them and handle the file transfer for you.

That's it! No, really, you don't need to do more to send files over AirDrop from your app!

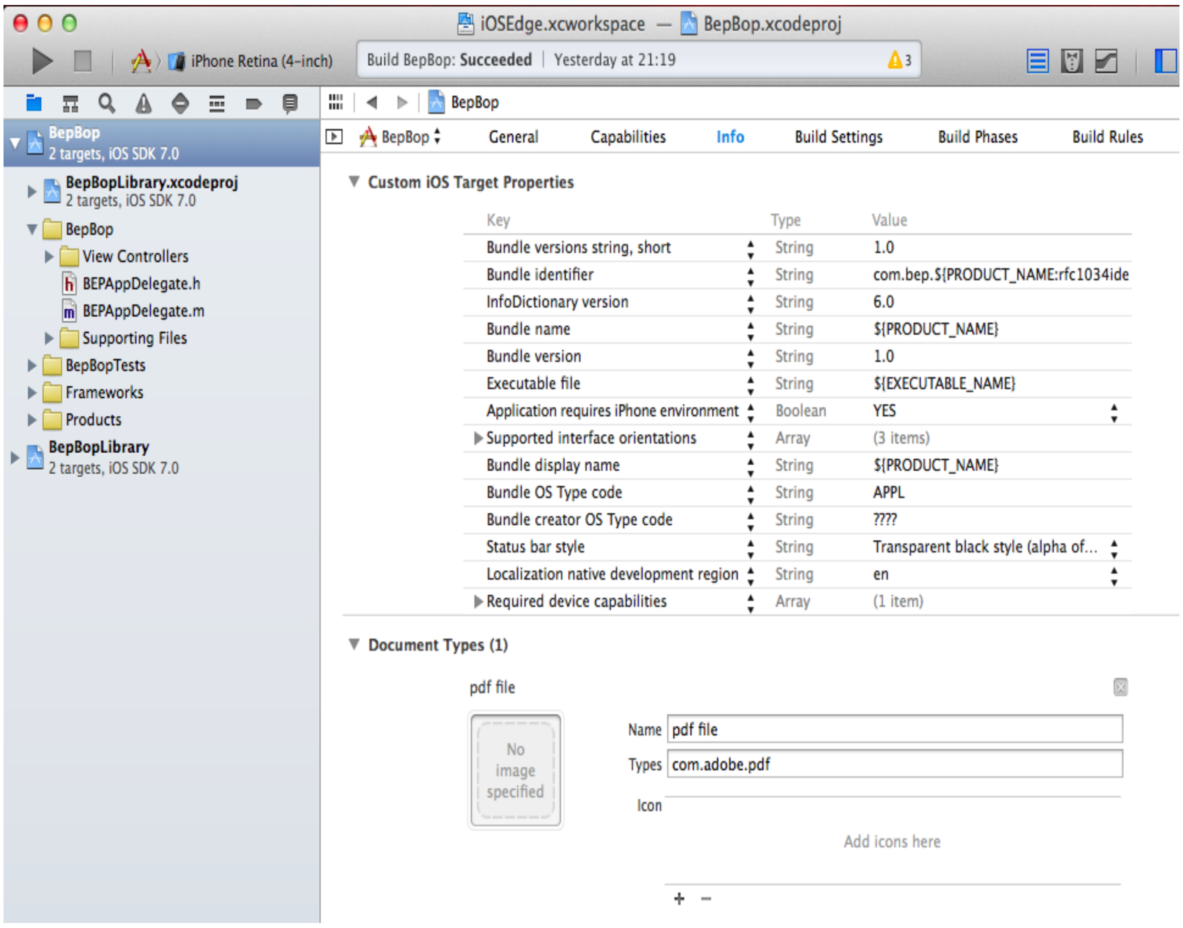
Receiving Files over AirDrop

Sending files over AirDrop was unbelievably easy, but accepting files over AirDrop and correctly handling them is a little bit more involved but nothing too complicated. There are two steps: First you need to declare your app as an handler to the files types you're capable of handling. And second once the system notifies your app of the reception of such a file you need to take it and copy it to a local directory from its temporary AirDrop location. Let's look at how we have done it for the BepBop app:

Declaring Your App as a Handler for a File Type

Apple uses the universal type identifiers (UTIs) to decide which resources can be handled by which apps. You can go readup on the details of the UTIs but for our purposes now UTIs are strings like public.image or com.adobe.pdf, which identify different formats of data and the system decides on a particular UTI for an item by looking at the MIME-type or the file extension.

If we want our app to be recognized as a handler for a certain file type we have to declare in as such by adding the corresponding UTI in our Info.plist. In Xcode 5 this can be achieved by going to the Info tab under our build target and adding our desired UTI under "Document Types". Here is an example of how we declared the BepBop.app to a handler for PDF files:



Custom iOS Target Properties

Key	Type	Value
Bundle versions string, short	String	1.0
Bundle identifier	String	com.bep.${PRODUCT_NAME:rfc1034ide
InfoDictionary version	String	6.0
Bundle name	String	${PRODUCT_NAME}
Bundle version	String	1.0
Executable file	String	${EXECUTABLE_NAME}
Application requires iPhone environment	Boolean	YES
▶ Supported interface orientations	Array	(3 items)
Bundle display name	String	${PRODUCT_NAME}
Bundle OS Type code	String	APPL
Bundle creator OS Type code	String	????
Status bar style	String	Transparent black style (alpha of...
Localization native development region	String	en
▶ Required device capabilities	Array	(1 item)

Document Types (1)

pdf file

No image specified

Name: pdf file

Types: com.adobe.pdf

Icon

Add icons here

One undocumented fact to keep in mind while deciding on which UTIs to handle with your app is that Apple does not treat all UTIs equally. Some UTIs such as public.image, public.jpeg and public.png are private and can only be handled by the default Photos.app on your iOS devices. This was probably a decision on Apple's part to protect users from being overwhelmed by all the photo apps which would complete to open their photos every time they received a photo over AirDrop.

Receiving Files Through Airdrop in Your App Delegate

You need to implement the method application:openURL:sourceApplication:annotation: in your application delegate to be able to receive files through AirDrop. When the system routes an AirDrop file to your app, this method will get called and the url will point to a file in your app's /Documents/Inbox directory. This directory is a special directory where your app has read permission but is unable to write to. If you need to modify the file you must move it to another directory first. Another point to pay attention to is, that the files in this directory can be encrypted using data protection. In the normal case the files inside your apps documents directory are freely available but in some cases, by the time this method gets called the user could have already locked the device. Therefore it is a good idea to check if the file is readable first by using the protectedDataAvailable property of the application object. Apple recommends to save the AirDrop URL for later and returning YES from this method even if the file could not be accessed due to data protection.

This method is a general purpose delegate method which gets called whenever the system tells your app to open a URL. Before iOS 7, apps used this functionality to pass data between apps by declaring their custom URL protocol and their own little API, like twitter://post?message=hello%20world&in_reply_to_status_id=12345. Starting with iOS 7 this method is also used by the system to tell your app that there is an incoming AirDrop file. To be able to distinguish between the opening a normal URL and receiving an AirDrop file cases, we can make use of several pieces of information:

- In the case of AirDrop sourceApplication is nil
- In the case of AirDrop url always contains /Documents/Inbox
- AirDrop url protocol will always be file://

Here is how we implemented the application:openURL:sourceApplication:annotation: method in the BepBop.app:

```
- (BOOL)application:(UIApplication *)application openURL:(NSURL *)url sourceApplication:(NSString *)sourceApplication
annotation:(id)annotation
{
 if (sourceApplication == nil &&
 [[url absoluteString] rangeOfString:@"Documents/Inbox"].location != NSNotFound) // Incoming AirDrop
 {
 NSLog(@"%@ sent from %@ with annotation %@", url, sourceApplication, [annotation description]);
 if (application.protectedDataAvailable) {
 [[BEPAirDropHandler sharedInstance] moveToLocalDirectoryAirDropURL:url];
 }
 else {
 [[BEPAirDropHandler sharedInstance] saveAirDropURL:url];
 }
 return YES;
 }

 return NO;
}
```

As you can see from the code above, it is quite straightforward to handle an incoming AirDrop file. If we have access to protected data we call the moveToLocalDirectoryAirDRopURL: method, which moves the file given to us by AirDrop to a local directory where we can modify the file as we see fit:

```
- (BOOL)moveToLocalDirectoryAirDropURL:(NSURL *)url
{
 NSFileManager * fm = [NSFileManager defaultManager];

 NSURL * localUrl = [[self airDropDocumentsDirectory] URLByAppendingPathComponent:[url lastPathComponent]];
 NSError * error;
 BOOL ret = [fm moveItemAtURL:url
 toURL:localUrl
 error:&error];

 if (!ret) {
 NSLog(@"could not move file from %@ to %@: %@", url, localUrl, error);
 }

 return ret;
}

```

On the other hand if protected data is not available for the time being because the device is locked with a passcode, we fallback to saving the incoming file URL in user defaults for later processing:

```
- (void)saveAirDropURL:(NSURL *)url
{
 NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
 NSArray * currentStoredURLs = [defaults arrayForKey:BEPDefaultsStoredAirDropURLsKey];

 if (currentStoredURLs == nil) {
 currentStoredURLs = @[];
 }

 NSArray * updatedStoredURLs = [currentStoredURLs arrayByAddingObject:url];

 [defaults setObject:updatedStoredURLs forKey:BEPDefaultsStoredAirDropURLsKey];

 [defaults synchronize];
}
```

The best way to process these saved URLs later is to add an observer to the UIApplicationProtectedDataDidBecomeAvailable notification and in the handler call the moveToLocalDirectoryAirDropURL: for each element of the saved URLs array.

The AirDrop feature is quite powerful and easy to use but does not provide an API to integrate more tightly in our apps. Luckily, however, Apple also introduced a new framework in iOS7, which enables us to build on top of the same technologies as AirDrop. Let's see how we can use it in the next section.

Multipeer Connectivity

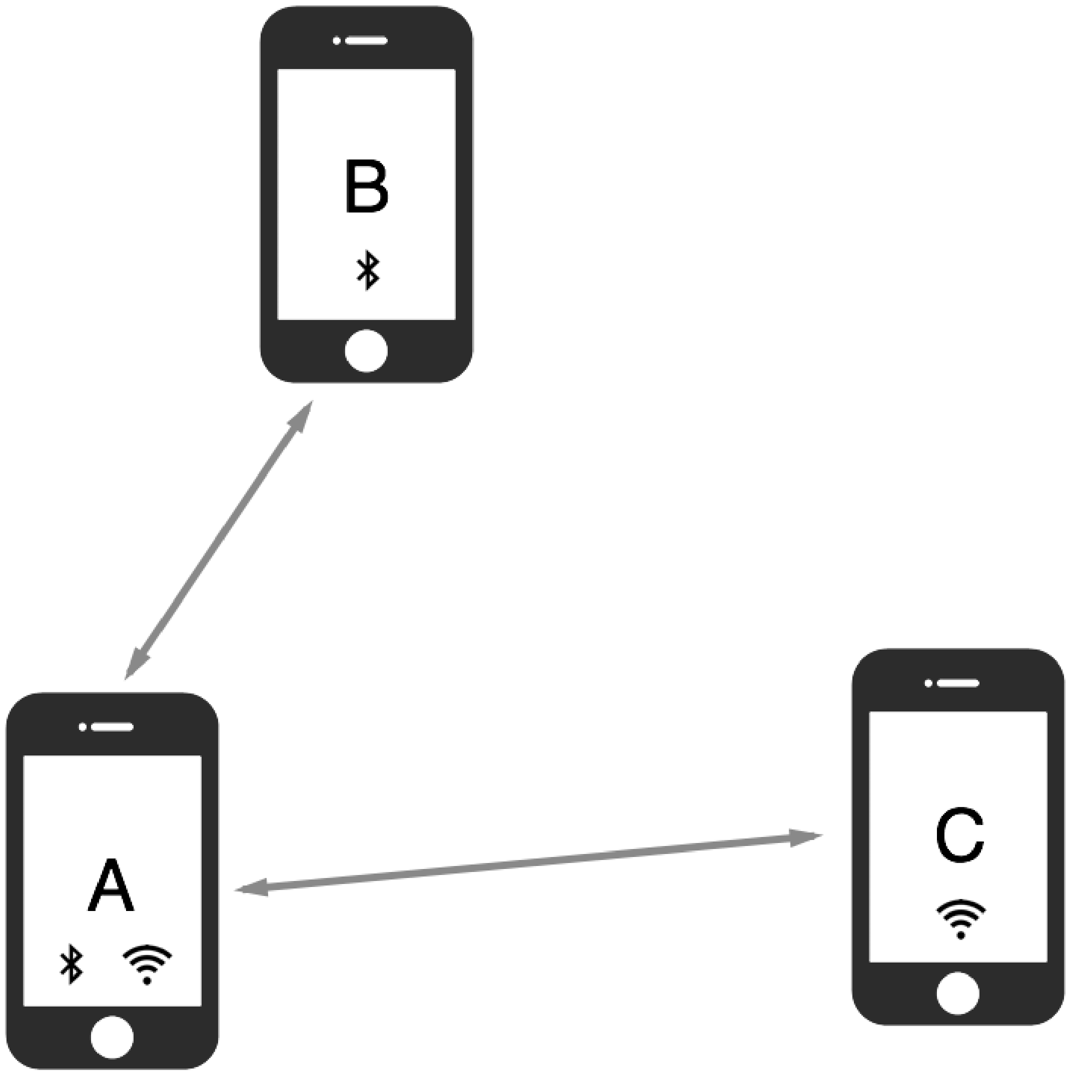
In iOS 7, Apple added a completely new framework called Multipeer Connectivity. This framework provides an easy interface to the powerful wireless technology underlying AirDrop and enables peer-to-peer wireless data transfer over bluetooth and adhoc Wi-Fi among groups of peers. iOS developers could build completely new and amazing apps using this new capabilities.

Technical Details

The Multipeer Connectivity (MC) framework enables both the discovery of services provided by nearby iOS devices and transfer of data between peers over infrastructure Wi-Fi networks, adhoc Wi-Fi and Bluetooth personal area networks. The low-level and wireless technology depended complexities are beautifully abstracted away so that the we, as app developers, can concentrate on developing new apps and features using the new services and functionality provided by the MC framework.

In the discovery phase, the MC-framework uses Apple's Bonjour technology to advertise and discover services and subsequently configure connections.

During data transfer the MC-framework intelligently uses the available wireless networks to make sure that the data is received by all peers in the session.



In the figure above, Peers A, B and C are in the same session. A has both Wi-Fi and Bluetooth interfaces available where as B and C only have Bluetooth and Wi-Fi respectively. In such a configuration when A sends data to the session, both B and C receive the data through their available interfaces. However since B cannot communicate with C directly due to the lack of a common interface, the MC-framework ensures that the data sent by B are transparently relayed via A to C such that every peer in the session can communicate with the other peers independent of the underlying network technology.

Basic Usage

Before you can start exchanging data between the Peers you first need to setup the session. This involves advertising your service and browsing for other peers. You see it action in our demo app BepBop. We will walk through the details of this initial phase of the setup in the next couple of sections and see how it all works together to give the user a seamless and easy connection experience.

How to Advertise Your Service – MCAdvertiserAssistant

Before a session can be joined it must be initialized and advertised so that other peers can find and join it. The most straight-forward way to do this is by using an advertising assistant. An advertising assistant sits on top of the MCNearbyServiceAdvertiser, which advertises our service over the network, and takes care of various tasks such as showing invitation confirmations and such. We do this by initializing an instance of the class MCPeerID, which will identify our device for the purposes of MC-framework and we are responsible for keeping it unique in our apps:

```
NSString* deviceName = [[UIDevice currentDevice] name];
self.myPeerID = [[MCPeerID alloc] initWithDisplayName:deviceName];
```

Then we can initialize an empty session with our peer ID and assign a delegate to it. In the example code we will be implementing the MCSessionDelegate protocol in our view controller and can thus assign self as delegate:

```
self.session = [[MCSession alloc] initWithPeer:_myPeerID];
_session.delegate = self;
```

Now we're ready to setup our advertiser assistant by initialzing an instance of MCAdvertiserAssistant with our service type and session. Since the advertiser assistant also needs a delegate to inform us of received invitations, we will implement MCAdvertiserAssistantDelegate in our view controller and assign self also as the delegate to our advertiser assistant and finally start the assistant:

```
self.assistant = [[MCAdvertiserAssistant alloc] initWithServiceType:BEPMultipeerConnectivityServiceType
discoveryInfo:nil session:_session];
_assistant.delegate = self;
[_assistant start];

```

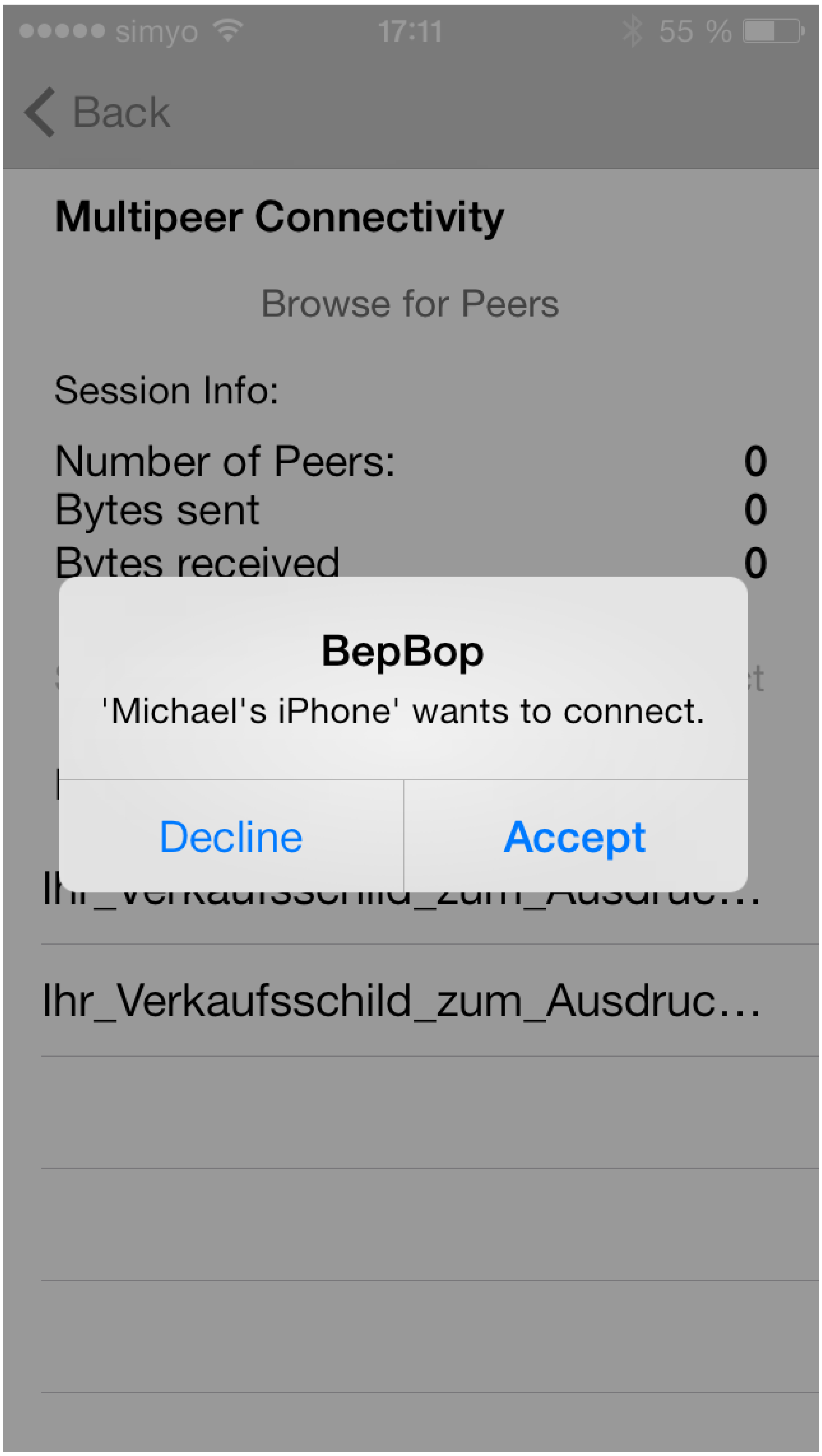
Now that our service is running and being advertised over the network, it is a good time discuss the details of the two delegate protocols we need to implement. MCSessionDelegate protocol is used to inform us of changes in session membership and data reception. We will go into the details of data reception in the following sections but for the purposes of advertising a service the most important method in the session delegate is session:peer:didChangeState: which tells us if peers have succesfully joined our session or not:

```
- (void) session:(MCSession*)session peer:(MCPeerID*)peerID didChangeState:(MCSessionState)state
{
 dispatch_async(dispatch_get_main_queue(), ^{
 [self updateUIState];
 });
}

```

The other protocol we just claimed to implement (by providing empty implementations), MCAdvertiserAssistantDelegate, allow us to intercept just before an invitation confirmation will be shown and just after it was dismissed. The actual process of receiving invitations, showing confirmations for them and sending out acceptances or rejections appropriately is taken care of by the advertiser assistant itself (in the following sections we will cover the details of how to programmatically handle invitations with the MCNearbyServiceAdvertiser and its corresponding delegate).

The advertiser assistant shows the following confirmation when it receives an invitation to connect. You can either accept to let the peer join the advertised session or reject.



Multipeer Connectivity

Browse for Peers

Session Info:

Number of Peers: 0
Bytes sent 0
Bytes received 0

BepBop

'Michael's iPhone' wants to connect.

Ihr_Verkaufsschild_zum_Ausdruc…

Ihr_Verkaufsschild_zum_Ausdruc…

How to Browse for Services – MCBrowserViewController

Now that we covered how to advertise our service, let's look at how other peers can discover it and join its session. MCBrowserViewController is the easy to use standard way of browsing for specific service types, which does most of the heavy lifting associated with finding peers and sending out invitations. First we need to initialize our peer ID and an empty session, just like we did for the MCAdvertiserAssistant, before we can initialize a MCBrowserViewController with the service type we want to browse for. Once we have the browser, we assign a delegate implementing MCBrowserViewControllerDelegate and present it on our view controller:

```
- (IBAction) browseButtonTapped:(UIButton*)sender
{
 NSAssert(_session != nil, @"trying to start browser but our session is nil");
 self.browser = [[MCBrowserViewController alloc] initWithServiceType:BEPMultipeerConnectivityServiceType
 session:_session];
 _browser.delegate = self;
 [self presentViewController:_browser
 animated:YES
 completion:nil];
}
```

The browser shows a list of nearby peers advertising the requested service type. Tapping one of the peers sends them an invitation that they can either accept or decline. The invitation could either be handled programmatically by the other peer (discussed in the following sections) or shown by an advertiser assistant (see above).

Sending Data Between Peers

Once we have other peers in a session we can start sending and receiving data. The MC-framework abstracts away all network-level functionalities and provides an easy-to-use API in the form of MCSession and MCSessionDelegate, allowing sending and receiving of three types of objects:

1. Messages as NSData instances
2. Resources as NSURL instances
3. Data streams as NSStream instances

Let's quickly introduce the methods involved in each case.

Sending Messages

Messages are sent by a MCSession instance's sendData:toPeers:withMode:error: method and handled by the delegate's session:didReceiveData:fromPeer: method at the other end. Even though there is no explicit limit on the size of data that can be sent using these methods, the best practice is to use these methods only for small messages because they don't provide any way to track the progress of the transfer. Let's look at how we send an example dictionary when the user taps the "Say Hello" button in the demo app:

```
 NSDictionary * message = @{@"message": @"Hello World!"};

 NSString * errorString = nil;
 NSData * data = [NSPropertyListSerialization dataFromPropertyList:message
 format:NSPropertyListXMLFormat_v1_0
 errorDescription:&errorString];
 if (errorString) {
 NSLog(@"could not serialize\n%@\n%@", message, errorString);
 }
 else {
 NSError * error;
 if (![_session sendData:data
 toPeers:[_session connectedPeers]
 withMode:MCSessionSendDataReliable
 error:&error])
 {
 NSLog(@"failed sending data");
 }
 }
```

The delegate parses the received data as follows:

```
 NSPropertyListFormat fmt;
 NSError * error;
 NSDictionary * message = [NSPropertyListSerialization propertyListWithData:data
 options:NSPropertyListImmutable
 format:&fmt
 error:&error];
 if (message == nil) {
 NSLog(@"could not parse received data: %@\n%@", error, data);
 return;
 }
```

Because we sent with the reliable mode using the MCSessionSendDataReliable parameter, the MC-framework will automatically retry in the case of dropped packages and guarantee an in order delivery using send and receive buffers. This mode functions similar to TCP protocol and while making it straight-forward to reliable transfer data is not suitable for multi-media data where latency is more critical than reliability. Low-latency, non-reliable mode which pushes data immediately can be selected by using the MCSessionSendDataUnreliable parameter.

Sending Resources

If you need to transfer larger files, your best solution is to use the sendResourceAtURL:withName:toPeer:withCompletionHandler: method in MCSession. When given a file URL this method will reliably send that file to the specified peer and call its completion handler once the transfer is complete. In the following code snipplet you can see how the BepBop app uses the resource sending functionality of MCSession to transmit a photo to another peer:

```
NSURL * tempURL = // write image data to a temporary location

// the completion handler which shows either a success or error message
id completionHandler = ^(NSError* error){
if (error) {
// Show failure alert
}
else {
// Show success alert
}
};
//send the temp file to all connected peers in session.
for (MCPeerID * peer in _session.connectedPeers) {
[_session sendResourceAtURL:tempURL
withName:[tempURL lastPathComponent]
toPeer:peer
withCompletionHandler:completionHandler];

}
```

We have removed the tedious part of the code to emphasize the important parts. Please note that unlike sending messages, resources cannot be sent to multiple peers at once. That's why we're looping through all connected peers in session. The completion handler is a block gets passed an error parameter once sending is completed. If the error parameter is nil the transmission was completed successfully, the error object will otherwise contain information about what went wrong.

On the other end, the delegate method session:didStartReceivingResourceWithName:fromPeer:withProgress: will be called when the transfer starts. This is a good place to initialize the UI which shows the status of the file transfer using the progress object. Once the transfer is complete the delegate method session:didFinishReceivingResourceWithName:fromPeer:atURL:withError: is called. If there are no errors the received file is located at a temporary location and the receiver is responsible to copy the received file to the desired location.

Sending Streams

If you need a low level stream to another peer to pipe data as it becomes available, you should look into the startStreamWithName:toPeer:error: of MCSession and the corresponding session:didReceiveStream:withName:fromPeer: method of the MCSessionDelegate. Once the streams have been established they can be used as any other output or input stream by the sender and respectively the receiver. We will not go into further detail here since most apps will not need such low-level network streams functionality. Detailed info can be found in Apple documentation.

Advanced Cases

We have already seen how to advertise and browse for services using the provided convenience classes MCAdvertiserAssistant and MCBrowserViewController, which also take care of the invitation process for us. The MC-framework also offers the possibility to programmatically control the advertising and browsing processes to obtain more flexibility and offers the possibility to build a custom browsing UI. Moreover, authentification of peers and encryption of transmitted data is also supported. Let's see how these work in the next three sections.

Programmatically Advertising and Handling Invitations

If you need more control on the advertising and invitation processes, the MCNearbyServiceAdvertiser and the corresponding delegate protocol MCNearbyServiceAdvertiserDelegate provide deeper programmatic control. The biggest difference to the advertiser assistant is that we are responsible for adding and removing peers to the session, and handling invitations but we have more options when advertising the service. An instance of MCNearbyServiceAdvertiser is initialized using initWithPeer:discoveryInfo:serviceType:, where discovery info is a dictionary of key-value pairs, which is attached to our advertisement as a Bonjour TXT-record. This provides an extra channel where can put some information for the peers browsing for our service type:

The corresponding MCNearbyServiceAdvertiserDelegate protocol defines the advertiser:didReceiveInvitationFromPeer:withContext:invitationHandler: method which gives us complete control on how to handle an incoming invitation. Whereas the MCAdvertiserAssistant shows a UIAlert to let the user accept or reject the invitation, using this delegate method we have the possibility to come up with ways to allow automatic acceptance or rejection of peers based on our app's criteria.

Programmatically Browsing for Peers

Similarly to MCNearbyServiceAdvertiser, MCNearbyServiceBrowser provides complete control on the browsing process and gives a foundation to build a custom browsing UI through it delegate protocol MCNearbyServiceBrowserDelegate and its methods browser:foundPeer:withDiscoveryInfo: and browser:lostPeer:

The MCNearbyServiceBrowser method invitePeer:toSession:withContext:timeout: is used to invite other peers into a session allows passing auxiliary data with the context parameter to advertiser:didReceiveInvitationFromPeer:withContext:invitationHandler: method of the advertiser delegate. The outline of such an automated algorithm could look like the this:

- Browser peer invites the other peer with a key as context data

```
 [browser invitePeer:advertiserPeer
 toSession:_mySession
 withContext:@{@"key": myBase64EncodedKey}
 timeout:60];

```

- Advertiser delegate on the other peer checks if this key is allowed

```
- (void) advertiser:(MCNearbyServiceAdvertiser *)advertiser
 didReceiveInvitationFromPeer:(MCPeerID *)peerID
 withContext:(NSData *)context
 invitationHandler:(void (^)(BOOL, MCSession *))invitationHandler
{
 NSString * base64EncodedKey = context[@"key"];

 BOOL keyValid;

 // Check if the provided key is valid

 if (keyValid) {
 invitationHandler(YES, _mySession);
 }
 else {
 invitationHandler(NO, nil);
 }
}
```

Authentication and Encryption

In addition to the powerful discovery and data transfer features, the MC-framework also supports industry standard authentication and encryption methods. A secure session can be created like this:

```
MCSession *session = [[MCSession alloc]
 initWithPeer:myPeerID
 securityIdentity:identity
 encryptionPreference:preference];
```

where identity is an array containing a SecIdentityRef object and (optionally) additional certificates needed to verify that identity and preference is one of MCEncryptionNone, MCEncryptionOptional or MCEncryptionRequired. An unencrypted session can only interact with unencrypted peers and a session with mandatory encryption can only interact with encrypted peer whereas a session with optional encryption can accept both encrypted and unencrypted peers.

When a peer tries to establish connection to a secured session, the session delegate method session:didReceiveCertificate:fromPeer:certificateHandler: will be called. It is the session delegates responsibility to validate the provided chain of certificates and either accept or reject the connection.

Summary

In this chapter we have introduced the new AirDrop feature in iOS7 and covered how you can add AirDrop support to your apps. Then we introduced the Multipeer Connectivity framework, on top which the AirDrop feature is built. Using the MC-framework it is possible to easily create apps, which communicate with their nearby peers to enable previously impossible new features. We first walked through how advertise and discover services using the high-level classes in the framework. Then showed how you can send and receive data once the connection has been established between the peers. And finally moved on to the more advanced features of the framework such as custom advertisement and invitation methods, how to provide a custom browsing UI and finally how to setup secure sessions.

6. Map Directions in 3D

IN THIS CHAPTER

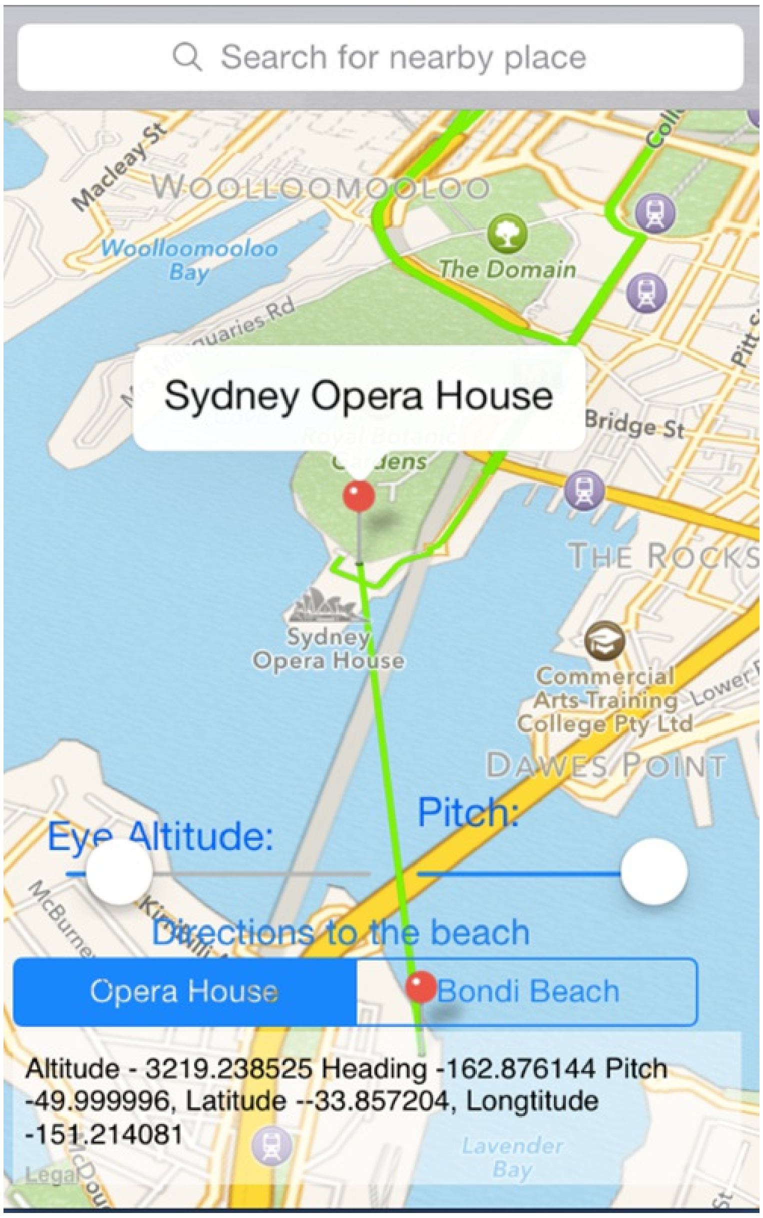
- User Interface Changes
- 180th Meridian Span
- Overlays
- Local Search
- Directions
- Perspectives
- Saving and Restoring Map State

Moving on from the previous chapter which explored the topic of *AirDrop* and *Multi-Peer Connectivity*, this chapter will focus on the new improvements on the ever-maturing *Apple Maps*, in its second iteration since Apple's decision to build their own solution from the ground-up. Granted, Apple Maps received a lot of negative press since it came out last year, due to incomplete data compared to Google Maps, but no one could deny that visually, the cartography with fly-overs looked amazing.

In iOS 7, it is quite obvious Apple has spent a great deal of resources in not only ironing out missing geospatial data, but improving the UI markedly, first in order to get it visually in-line with the iOS 7 theme, but also with subtle improvements in the cartography. MapKit, the API that powers the maps has also received some great updates.

MapKit changes in iOS 7 include some eye-candy 3D perspectives, allowing developers to control camera pitch and position programmatically, as well as some overlay improvements, better direction engine logic as well as some other nifty things, such as Directions, which will be examined in this chapter.

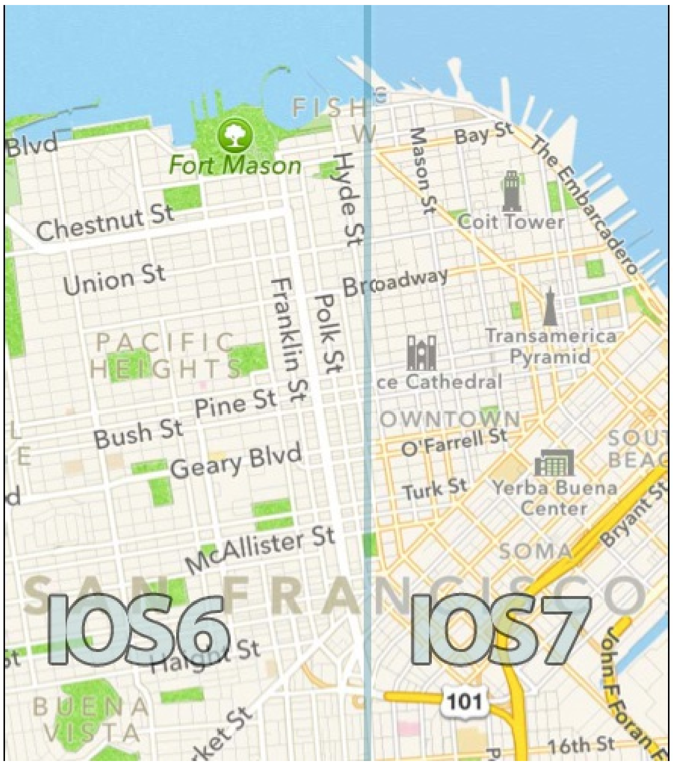
We are continuing on with our BepBop app, which you can find in the Chapter 9 section of the sample code. In this chapter, we will be demonstrating many of the features available in iOS 7, including setting annotations, manipulating the 3D camera perspective, searching using natural query language, working with overlay poly-lines, and getting directions to a particular location. You can compile and run the BepBop project and go to Chapter 9 section, and you will see a screen resembling the one below, which you can use to test the various new features of MapKit, along with the sample code for the chapter:



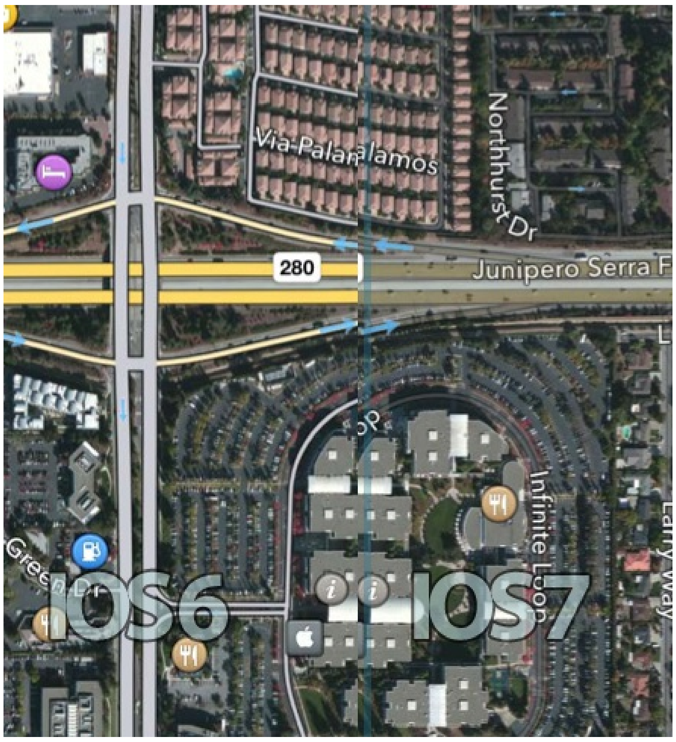
Altitude - 3219.238525 Heading -162.876144 Pitch -49.999996, Latitude --33.857204, Longtitude -151.214081

User Interface changes

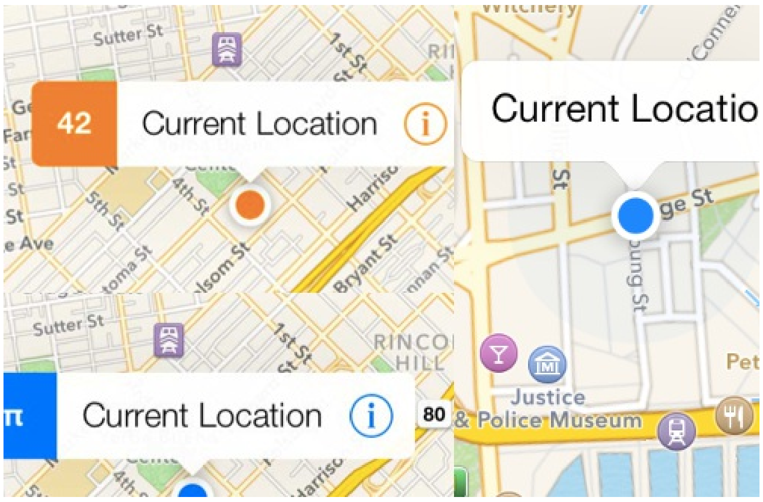
In iOS 7, Maps was visually refreshed in line with the rest of Apple's UI, following the *flattening* trend, which included predominant cartographic amendments. These are eye-candy improvements, but do provide more detail, crisper data, in both normal mapping and hybrid mapping views.



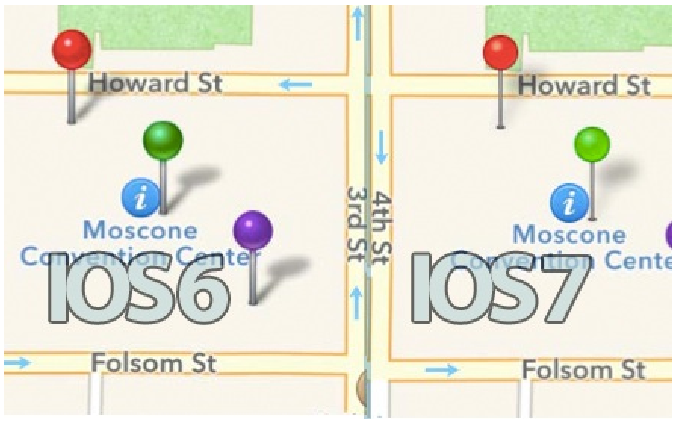
In both normal and hybrid map perspectives, the roads and labels have been better defined, to make reading them more clear.



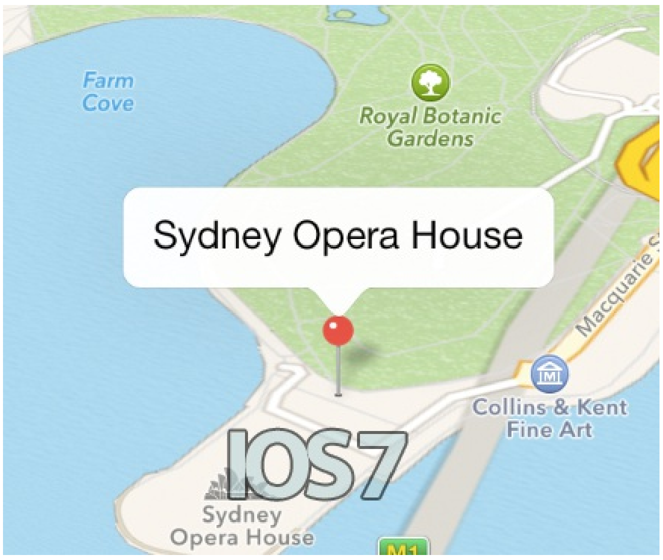
The annotation pins, callouts and user location indicators have also been flattened, with the depth in gradients removed and cleaned out. The *tintColor* is a new property that allows you to set a tinting theme to the callout and user-location dot.



The next illustrationcompares how the MKAnnotationPins have been flattened iOS 7, subtly.



Even when you pitch the map in 3D, the MKAnnotation pins retain the same dimensions, so as a developer you don't have to worry about any pin distortions if you implement your own custom pin view.



The best part of migrating from iOS 6 to iOS 7 is that so far you, the developer, don't need to do anything extra. The existing pin dimensions and aspect ratio remain the same as they did in iOS 6, so your assets won't get distorted when the map pans into 3D in any angle. One new API as far as annotations goes is a convenience method to show more than one pin easily:

```
[self.mapView showAnnotations:@[pointOperaHouse, pointBondiBeach] animated:YES];
```

This is certainly more convenient than previously having to individually add an annotation:

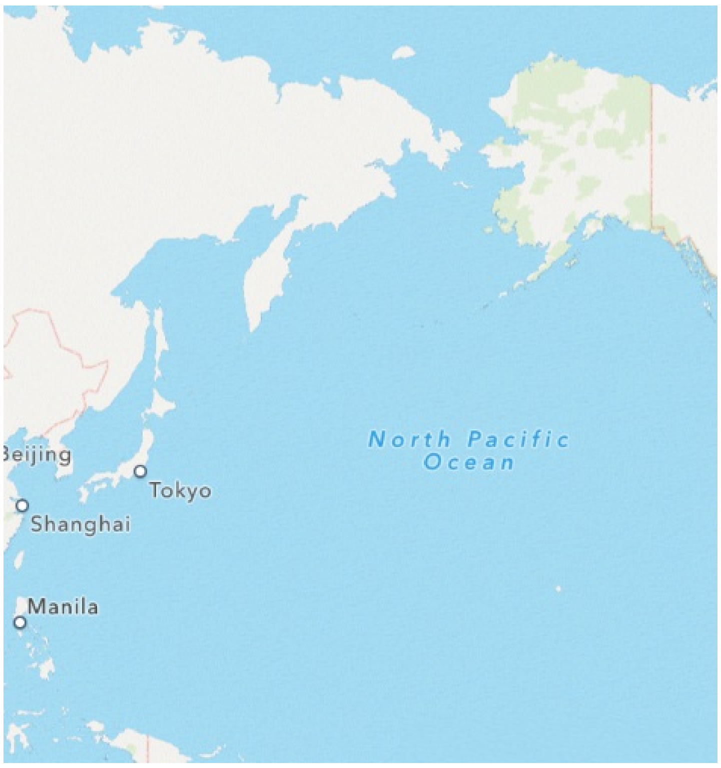
```
[self.mapView addAnnotation:point];
```

Another improvement from Apple requires no API changes, and comes pre-baked, the ability to cross the 180th meridian span, either via touch or programmatically, as the next section will outline.

180th Meridian Span

While Apple's Maps app was able to scroll the entire map across all meridians, developers were not able to replicate that easily in their own apps. New to iOS 7, without any extra coding requirements developers are now able to pan from the west coast of the U.S to Sydney, across the 180th meridian. The implications being, you can now set two points, one in Los Angeles and another across the anti-meridian, such as Sydney, and drawing a geodesic polyline will show you the shortest path between these two points, which is across the Pacific ocean. Previously, it would work it's way across the Atlantic Ocean, Europe and Asia to make it's way down to Sydney.

> The **180th meridian** or **antimeridian** is the meridian which is 180° east or west of the Prime Meridian with which it forms a great circle. It is common to both east longitude and west longitude. It is used as the basis for the International Date Line, because it for the most part passes through the open waters of the Pacific Ocean. However, the meridian passes throughRussia and Fiji as well as Antarctica. -- Wikipedia (http://goo.gl/36UbVg)



This makes things easier for apps such as flight apps, where you want to show the shortest path between any location, and assert that you will always get the shortest route. You can test that theory by setting up a region or coordinate that is on the 180th meridian line, like:

```
self.mapView.centerCoordinate = CLLocationCoordinate2DMake (0, 180);
```

This sets the point at 0 latitude and 180th longitude. So, regardless of any point on the planet, you will now get the shortest route plotted. Speaking of drawing paths between two points, there are some improvements in overlays and geodesic polyline paths, newly introduced in iOS 7 which will now be examined.

Overlays

Overlays in iOS 7 Maps has been overhauled significantly, starting with API changes that require developers ensure in SDK 7 that ***MKOverlayView*** is replaced with **MKOverlayRenderer**, a more light-weight and robust reference that allows the developer to set overlays at specific levels in the map. The API works the same way as previously, but loses the ancenstry to UIView.

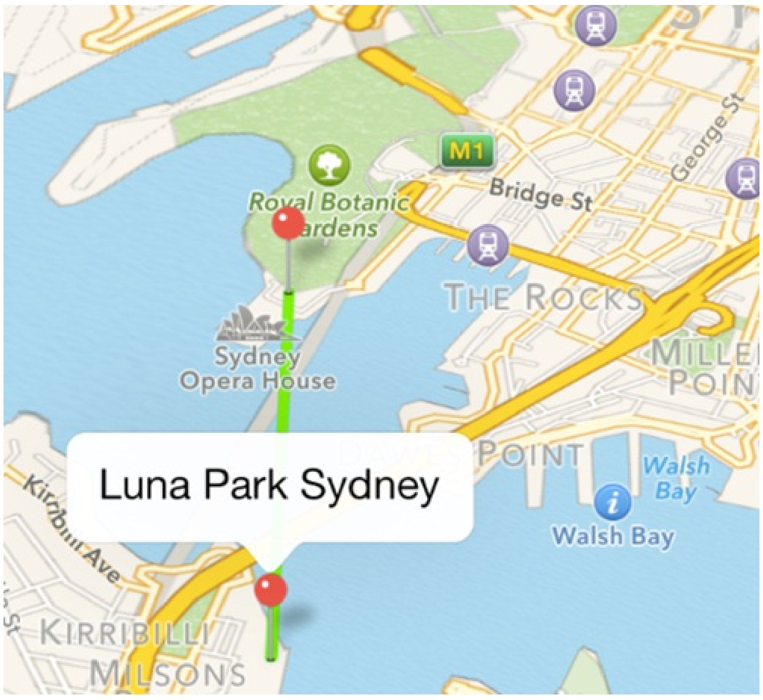
Overlays can now also be inserted at two levels:

1. **MKOverlayLevelAboveRoads** : sets the overlay *above the roads* but *beneath labels, POIs and Annotation Views*;
2. **MKOverlayLevelAboveLabels** : covers the entire map, including labels, but still affords you the ability to add annotation views visibly above the overlay.

Apple have also given us access to a new class, called **MKGeodesicPolyline** which is the shortest path between two points along a curved surface. Here is how we implemented a simple poly-line between two points in our BepBop.app:

```
CLLocationCoordinate2D points[] = {lunaParkCoordinate, sydneyOperaHouseCoordinate};
 geodesic = [MKGeodesicPolyline polylineWithCoordinates:points count:2];
 [self.mapView addOverlay:geodesic level:MKOverlayLevelAboveRoads];
```

The result is shown below, a simple geodesic line between two coordinates, *Sydney Opera House* and *Luna Park Sydney*. Note that this is just a direct poly-line and not using the directions API, which would present the route through various roads.



The next topic deals with *Local Search*, which in iOS 7 has introduced the ability to search for nearby places, using *natural query language*, like "Cafes" or "Cinema", which we will dive into next.

Local Search

Whilst we were previously able to perform searches using geo-coding and reverse-geocoding, iOS 7 has given us the ability to search contextually (locally) through natural language queries. This is done by firstly creating an object of type **MKLocalSearchRequest**, assigning a local search query string to the object, and constrainign the search locally to **newRegion**, which we passed in the Sydney Opera House region:

```
 MKLocalSearchRequest *request = [[MKLocalSearchRequest alloc] init];

 request.naturalLanguageQuery = searchString;
 request.region = newRegion;
```

We then set out the search completion handler, prior to initiating the search:

```
MKLocalSearchCompletionHandler completionHandler = ^(MKLocalSearchResponse *response, NSError *error)
{
if (error != nil)
{
//handle error
}
else
{
NSLog(@"response %@", response.mapItems);
self.places = [response mapItems];
[self addPlacesAnnotation];

}
...
};
```

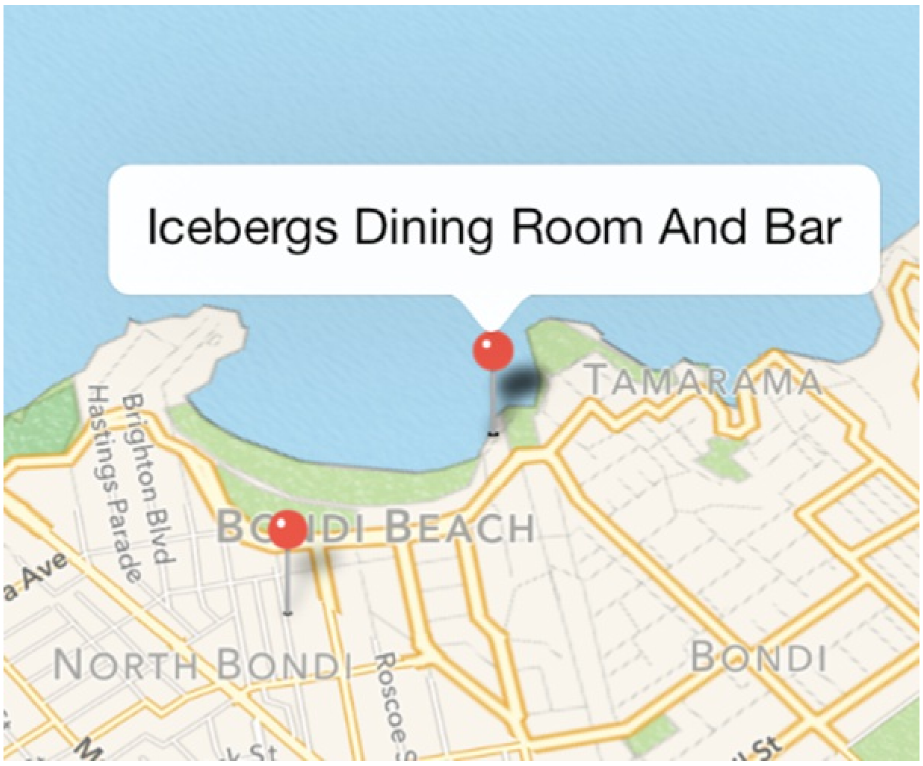
We get a set of **MKMapItem** back, which we then store in an array, before setting them as annotations in the map.

. . .

```
self.localSearch = [[MKLocalSearch alloc] initWithRequest:request];

[self.localSearch <strong>startWithCompletionHandler:completionHandler];</strong>
[UIApplication sharedApplication].networkActivityIndicatorVisible = YES;
```

We have an instance of **MKLocalSearch** that we use to call **startWithCompletionHandler**, passing in the handler block we just defined previously, so that it can run asynchronously. We searched using the query string "Icebergs", with result finding two locations, as illustrated in the figure below:



Once you've searched for a location, wouldn't it be nice to get directions to it? The next section will show you how to make use of the new *routing* API to get the best route between two points.

Directions

In iOS 7, Apple has given developers greater power through exposing a *routing API* that is truly empowering. With the latest API revisions, we are able to provide directions that:

- Adapt to either driving or walking modes of transport; Walking;
- Displaying alternative routes to the destination;
- Estimation of time to destination based on current traffic conditions;
- Estimation of future time to destination based on historical traffic data;

To use the routing API is also quite straightforward. First, you create an **MKDirectionsRequest** object, passing it a *source* and *destination* MKMapItem. You then set optional properties, such as whether you require alternative routing to be displayed, before calling the block method to get directions, as we had done in our BepBop app:

```
MKDirections *directions = [[MKDirections alloc] initWithRequest:request];
 [directions calculateDirectionsWithCompletionHandler:
 ^(MKDirectionsResponse *response, NSError *error) {
 if (error) {
 //handle error
 } else {
 [self displayDirectionsWithResponse:response];
 } }];
```

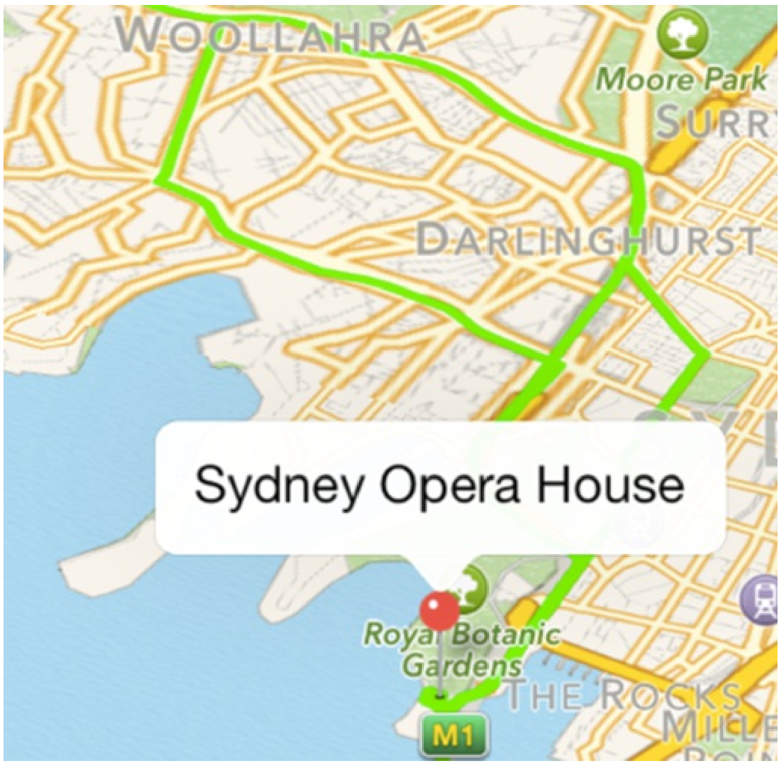
In calling the block, provided we have no errors come our way, we call the *displayDirectionsWithResponse:* method, passing in our **MKDirectionsResponse** object. When then, wrapped in another UIView Animation block, loop through all the routes in the response and add them as poly-line overlays in our map.

...

```
for (MKRoute *route in response.routes) {
 [self.mapView addOverlay:route.polyline level:MKOverlayLevelAboveRoads];
```

...

The following shows the two routes returned in the BepBop app, between the Sydney Opera House and Bondi Beach, taking into account traffic conditions.



A thing to note is that in the **MKDirectionsResponse** object returned, it echos the source and destination passed in the request back, to allow you to re-confirm that those are the locations originally intended. It may actually vary from the original request, due of various reasons, such as a more optimal point/path being chosen by the API as the starting or ending point.

The response also delivers other properties worth noting, such as:

- Localised Name of the route, such as "Route 49M";
- Distance of the journey;
- Localised advisory notices, such as "requires toll";
- Expected travel time;
- Geometry poly-line (which we use to draw on our map);
- Steps in the journey, also known as way-points;

As you can see, directions is quite easy to implement, with only a few lines of code. There currently isn't any application-capped or developer-capped usage limits, with Apple promising that popular apps won't be throttled because of their popularity. Having said that, Apple does recommend that fair-use apples, and Apple will throttle apps with extremely high usage, or emit high usage because of a bug.

The final section will dive into one of the most exciting parts of maps, working with 3D perspectives, and playing around with the 3D camera programmatically.

Perspectives

The most prominent visual change when Apple went with its own implementation of its mobile mapping platform was the feature of 3D buildings, and fly-overs, which certainly made Apple Maps one of the prettiest mapping apps, if anything. The ability to pitch and rotate and go across the entire meridian all existed in iOS 6, but not in MapKit, meaning in your own custom applications, you could not replicate the same wholesome features we had in Apple Maps.

When we talk about 3D maps, it isn't truly accurate that the map is 3D. In fact the cartography is 2D, so we have a 2D map, with 3D buildings that are extruded, giving the illusion of a complete 3D map, when in fact it is actually 2.5D, as shown in the following diagram:

Peet's Coffee & Tea
Yerba Buena Center For the Arts
Contemporary Jewish Museum
Mission St
3rd St
Martin Luther King Jr Memorial
Yerba Buena Center
Samovar Tea Lounge
Yogen Fruz @ Metreon
Metreon
Howard St
4th St
Jillian's
Legal

The user is able to use the same set of gestures to rotate and pan vertically the maps, with pinching to zoom that we are accustomed to in the Maps app itself. In fact, in XCode 5, you are even able to mimic the same gestures to test your app in the simulator, rotating using **option + drag** in a circular motion, whereas to pitch, **option + shift + drag vertically**.

As far as displaying 3D buildings, as well as pitching/rotating and zooming, that works right out of the box, when you compile your app in iOS 7. However, there are a few changes in the API that do require re-adapting, especially to take advantage of 3D, with the use of **MKMapCamera**.

Currently, you may be familiar with setting up a 2D simple map, without rotation, for instance like:

```
[MKMapView setVisibleMapRect:]
 [MKMapView setRegion:]
```

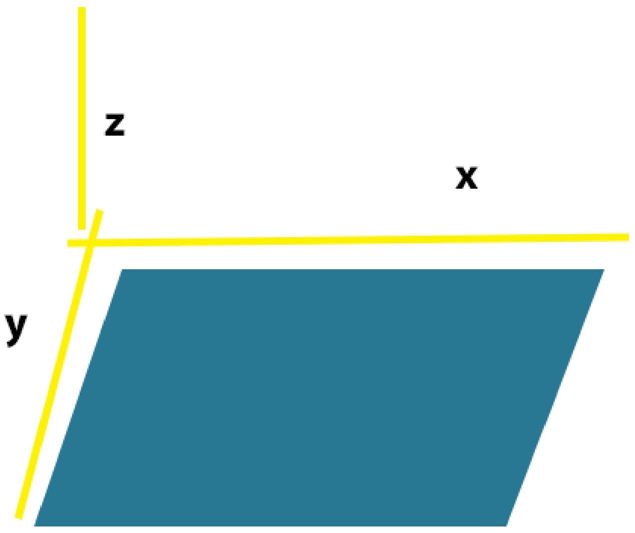
With 3D maps, because of the angle of the camera, the code aims ot get the best visible region taking into account the camera angle, based on a property called **centerCoordinate**, which is the center coordinate the camera should focus on. Whereas if we are viewing the map from the top, the center area is equally visible as a circle, but as we pitch the camera/angle, the area of visibility changes as well.

In previous iterations of iOS, we had MapKit expose four methods, to convert geometric properties, namely:

- ***convertPoint:toCoordinateFromView;***
- ***convertCoordinate:toPointToView;***
- ***convertRect:toRegionFromView;***
- ***convertRegion:toRectToView;***

The difference in iOS 7 is that when the user hits a point on the screen, because we are in 3D space, the user is now able to hit areas that are not physically on the map, such as beyond the horizon, towards the sky, where we are not really able to get coordinates for. Thus, the change is that now we can expect that it's possible to get invalid values as well, that may be returned, and this needs to be anticipated. This can in fact be anticipated with **kCLLocationCoordinate2DInvalid** and **CGRectNull**.

MKMapCamera is a new API class to iOS 7, that allows developers to play around with the perspective in 3D space. In order to understand how MKMapCamera works in the context of a 3D coordinate system, we must first understand that we have three axis to work with, the x, y and z.



The x and y axis form part of the 2D plane, whereas the z axis is the altitude, or third dimension. Looking at **MKMapCamera**, this correlates to the four important properties that help define the perspective:

1. Center Coordinate: which is the point on the ground as longitude and latitude, that is the center of the screen;
2. Altitude: refering to how high above the ground the camera should be;
3. Heading: which is the direction the camera should face, with 0 degrees being North, 180 degrees being South. This could conceptually be thought of as rotating along the Z-axis;
4. Pitch: is the angle the camera tilts at, with 0 being straight down, and the higher up from 0 in value, the more towards the horizon the camera points.

The last property, *pitch* in fact gets clamped or restricted to within a reasonable value, rather than allowing you to pitch all the way to the sky. The maximum value is different based on the other three properties, so the maximum is not always the same.

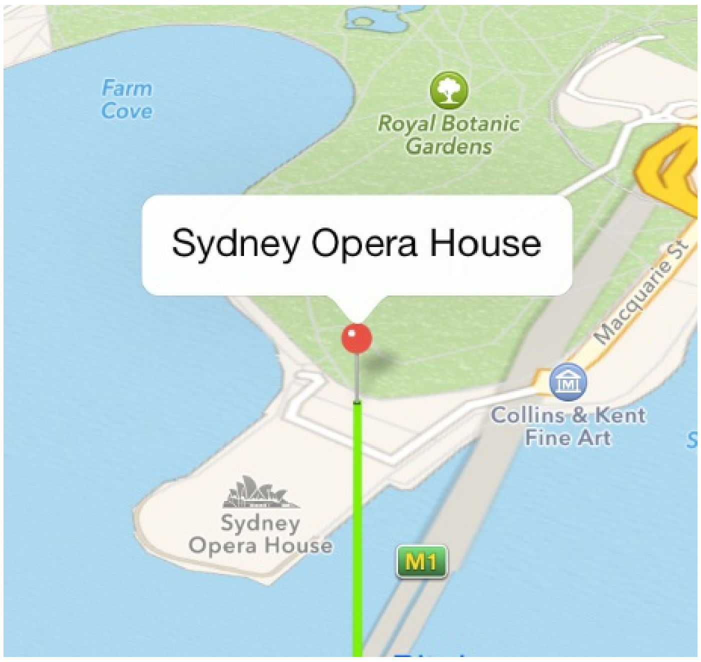
Using the **camera** convenience method is also non-trivial,and we will show you that as we demo it in our BepBop app:

1. In our ViewDidLoad: method, we set an **MKMapCamera** object, which will be our initial camera perspective. We set the lookingAtCoordinate to the specified Sydney Opera House coordinate, and set the fromEyeCoordinate: to be nearby landmark of Luna Park. We did that so we can get the viewing point, as if you are looking at the Opera House from Luna Park. We then set an eye altitude of 900 metres, so we can be hugh up, and set a pitch explicitly to 60.

```
MKMapCamera *camera = [MKMapCamera cameraLookingAtCenterCoordinate:sydneyOperaHouseRegion.center
fromEyeCoordinate:lunaParkRegion.center eyeAltitude:900];
 camera.pitch = 60;
 stepperValue = camera.pitch;

 //camera.altitude = 1500;
 [self.mapView setCamera:camera animated:NO];
```

The resulting perspective is illustrated below:



To demonstrate a fly-over effect in the BepBop app, we used multiple cameras to get from one location, *Sydney Opera House* to another location, *Bondi Beach.* We created a toggle button to switch between the two locations:

```
- (IBAction)changeCameraView:(id)sender {
 UISegmentedControl *segment=(UISegmentedControl*)sender;

 if (segment.selectedSegmentIndex == 0)
 [self goToCoordinate:sydneyOperaHouseCoordinate];
 else
 [self goToCoordinate:bondiBeachCoordinate];

}
```

We then created a method *goToCoordinate:* passing in the coordinate to animate to:

```
- (void)goToCoordinate:(CLLocationCoordinate2D)coordionate{
 MKMapCamera* end = [MKMapCamera cameraLookingAtCenterCoordinate:coordionate
 fromEyeCoordinate:coordionate
 eyeAltitude:500];
 end.pitch = 55;

 CLLocationCoordinate2D startingCoordinate = self.mapView.centerCoordinate;
 MKMapPoint startingPoint = MKMapPointForCoordinate(startingCoordinate);
 MKMapPoint endPoint = MKMapPointForCoordinate(end.centerCoordinate);

 MKMapPoint midPoint = MKMapPointMake(startingPoint.x + ((endPoint.y - startingPoint.y) / 2.0),
 startingPoint.y + ((endPoint.y - startingPoint.y) / 2.0)
 );

```

We created an end camera point, with the coordinates for the destination passed in as center coordinate as well as fromEye perspective, setting the altitude to be 500. We also set the end pitch to be 55. The starting Coordinate is the current location prior to the animation.

We then define the mid-point on the map to be half way between the starting and ending points. We also set the altitude distance zoom out four times, as shown below:

```
 CLLocationCoordinate2D midCoord = MKCoordinateForMapPoint(midPoint);

 CLLocationDistance midAltitude = end.altitude * 4; // zoom out 4 times

 MKMapCamera* midCamera = [MKMapCamera cameraLookingAtCenterCoordinate:end.centerCoordinate
 fromEyeCoordinate:midCoord
 eyeAltitude:midAltitude];

 listOfCameras = [[NSMutableArray alloc] init];

 [listOfCameras addObject:midCamera];
 [listOfCameras addObject:end];
 [self goToNextCamera];

}
```

We initialise a mutable array to store the list of Cameras (start, end and mid) and add the mid and end camera, before calling goToNextCamera:

```
- (void)goToNextCamera{
 if (listOfCameras.count ==0){
 return;
 }

 MKMapCamera* nextCamera = [listOfCameras firstObject];
 [listOfCameras removeObject:0];
 [UIView animateWithDuration:1.0
 delay:0.0
 options:UIViewAnimationOptionCurveEaseInOut
 animations:^{
 self.mapView.camera = nextCamera;
 } completion:NULL];

}
```

As long as we added cameras to the queue/stack we can continue with the method, getting the next camera and assigning it to our mapView, within a *UIView animateWithDuration:* block. So in three simple methods, we were able to re-created the fly-over effect that we see in Apple Maps, within our own mapping solution.

The final topic is certainly a pleasant attribute, the ability to archive and unarchive our map state using NSArchiver, and Unarchive, the same we we are normally accustomed to with other objects.

Saving and Restoring Map State

Saving and Restoring state is another thing the developer needs to bare in mind. When a user is pitched in a certain position and exits the app and gets back in, it would be nice to restore the state. With MKMapCamera, it inherits from **<NSSecureCoding>**, therefore you can follow the same archiving and de-archiving steps, we did in BepBop:

```
-(void)encodeRestorableStateWithCoder:(NSCoder *)coder{
 MKMapCamera* camera = [self.mapView camera];

 NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES);
 NSString *docDir = [paths objectAtIndex: 0];
 NSString* docFile = [docDir stringByAppendingPathComponent: @"BepBopMap.plist"];

 [NSKeyedArchiver archiveRootObject:camera toFile:docFile];

 [super encodeRestorableStateWithCoder:coder];
}

-  (void)decodeRestorableStateWithCoder:(NSCoder *)coder{

 NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES);
 NSString *docDir = [paths objectAtIndex: 0];
 NSString* docFile = [docDir stringByAppendingPathComponent: @"BepBopMap.plist"];

 MKMapCamera* camera = [NSKeyedUnarchiver unarchiveObjectWithFile: docFile];
 [self.mapView setCamera:camera];

 [super decodeRestorableStateWithCoder:coder];
}
```

The snippet above you should be accustomed to. The first method creating a directory path and file, **BepBopMap.plist** and archiving the current camera perspective to the file, whereas the second method unarchives the specified file accordingly and assigns it to the current mapView camera.

Summary

This chapter explored many of the new APIs introduced in MapKit, that allow you to leverage all the interactive and visually pleasing features found in Apple Maps. We introduced Overlays and new Local Search methods, using natural search query language, as well as the new Directions API to allow you to formulate the best routing path between two points.

We then walked through creating a 3D Camera animation, using MKMapCamera, programmatically adjusting the center coordinates, panning and pitching to customize your 3D viewing perspective. We concluded the chapter by outlining how we save and restore state for our mapping data objects, using the Archiving methodology.

In the next chapter we look at XCode 5 and the new *build improvements and continuous integration*, and how the latest IDE iteration saves you time.

7. Accessibility in iOS 7

IN THIS CHAPTER

- Why Accessibility?
- Dynamic Type
- Text to Speech

In this chapter, we'll talk about Apple's accessibility improvements to iOS 7 and some new APIs they've introduced. Designing for accessibility is essential to every app. The iPhone is designed with accessibility in mind, including technologies like VoiceOver to help those with visual impairments as well as settings for text size, screen zoom, and more, allowing those with low vision to better see and interact with your app's user interface.

Why Accessibility?

Apple has built Accessibility APIs in to iOS, and they make it really easy to support accessibility in your app. Not only should you do it because it's easy, but because it's right, it's profitable, and it's good for everyone - not just the disabled.

Stevie Wonder thanked Steve Jobs for making the iPhone accessible:

"I want you all to give a hand to someone that you know whose health is very bad at this time…. His company took the challenge in making his technology accessible to everyone. In the spirit of caring and moving the world forward: Steve Jobs. Because there's nothing on the iPhone or the iPad that you can do that I can't do. As a matter of fact, I can be talking to you, you can be looking at me, and I can be doing whatever I need to do and you don't even know what I'm doing. Yeah!"

Apple has included support for VoiceOver and accessibility APIs since iOS 3.0. In case you haven't looked at VoiceOver yet, here's a brief overview:

"Using iOS 3.0 and later, VoiceOver is available to help users with visual impairments use their iOS-based devices. The UI Accessibility programming interface, introduced in iOS 3.0, helps developers make their applications accessible to VoiceOver users. Briefly, VoiceOver describes an application's user interface and helps users navigate through the application's views and controls, using speech and sound." - Accessibility Programming Guide for iOS

The built-in support for VoiceOver in iOS makes it *ridiculously simple* for developers to make their apps easy to use for those with visual impairments. In iOS 7, Apple introduces new APIs that allow developers to make apps even more accessible, including Dynamic Type and Text to Speech.

Dynamic Type

While Dynamic Type isn't exactly part of Apple's new Accessibility APIs, it is new in iOS 7, and it does make iOS more accessible to those who have trouble seeing smaller text. If your apps support Dynamic Type, users can configure their preferred text size one time in the Settings app and have it apply to all apps that support it. This does require a bit of effort on your part as the developer, but we'll see that it's fairly simple and straightforward to do.

With Dynamic Type, your app can respond to the user's choice in text size. This is useful not only for users with low vision, but also for general users who just prefer a larger or smaller text size across all apps. Apple's built-in apps like Mail and Calendar support Dynamic Type, and your apps should, too.

When your app is set up to listen for changes in the user's preferred text size (Dynamic Type), you get automatic adjustments to weight, letter spacing, and line height for every font size.

You can listen for the UIContentSizeCategoryDidChangeNotification.

Dynamic Type give us six text styles to choose from. You get to choose from a Headline, Body, Subheadline, Footnote, and two types of Captions.

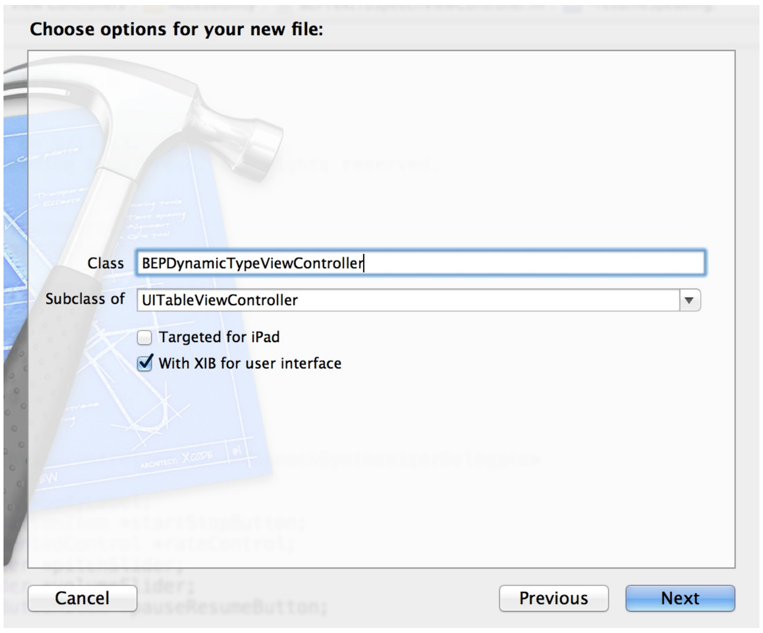
- UIFontTextStyleHeadline
- UIFontTextStyleBody
- UIFontTextStyleSubheadline
- UIFontTextStyleFootnote
- UIFontTextStyleCaption1
- UIFontTextStyleCaption2

For our BepBop project, we'll go with a headline, subheadline, body, and footnote. You'll find plenty of uses for these types in your own apps. Here, we're going to display an article from Apple's hot news feed.

Building a view that supports Dynamic Type

Let's create that view in Interface Builder. Start by creating a new view controller. We're calling ours BEPDynamicTypeViewController, but feel free to call yours whatever you'd like.

To create the new view controller, click File > New > File (or ⌘N). On the new screen, choose Cocoa Touch, Objective-C class, then click Next. On this screen, enter the name BEPDynamicTypeViewController, make it a subclass of UIViewController, check the box for With XIB for user interface, then click **Next**.


Choose options for your new file:

Class BEPDynamicTypeViewController

Subclass of UITableViewController

Targeted for iPad

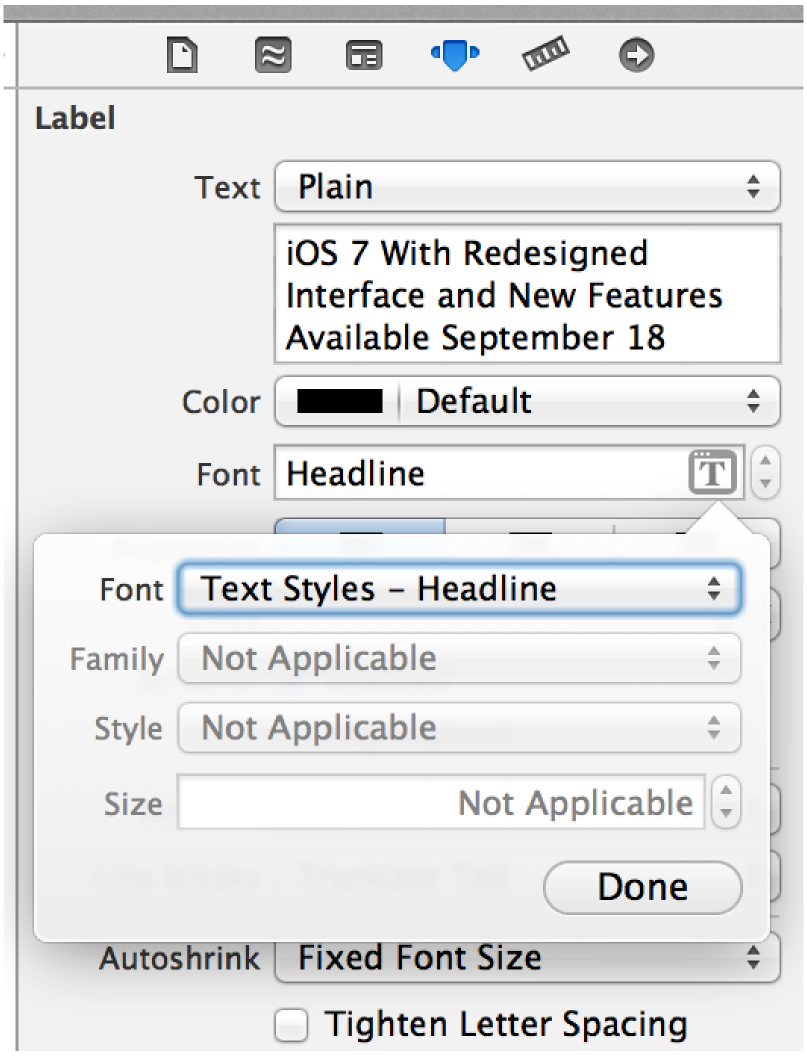
✓ With XIB for user interface

On the following screen, make sure it's included in the right group (BepBop, View Controllers, Accessibility) and that it's added to our BepBop target. Then click **Create**.

Now let's go to BEPDynamicTypeViewController.xib and build our view. We'll drag out four labels - one for the headline, one for the subhead where we'll show the date, one for the body of the article, and one for the footer that shows the "Read More" link. We'll also tell Interface Builder to simulate a navigation bar since we know that'll be there in our app.

The next step is to configure each label with text and the appropriate text style. In Interface Builder, this is quite simple. Let's start with the headline label:

1. Drag a label onto the view for the headline.
2. Double-click on it to set the text. We'll set the text to "iOS 7 With Redesigned Interface and New Features Available September 18".
3. With the label still selected, find the Font in the Attributes Inspector (in the Utilities pane on the right). Click the 'T' button, then set the font to Text Styles - Headline.


Label

Text: Plain

iOS 7 With Redesigned Interface and New Features Available September 18

Color: Default

Font: Headline

Font: Text Styles – Headline

Family: Not Applicable

Style: Not Applicable

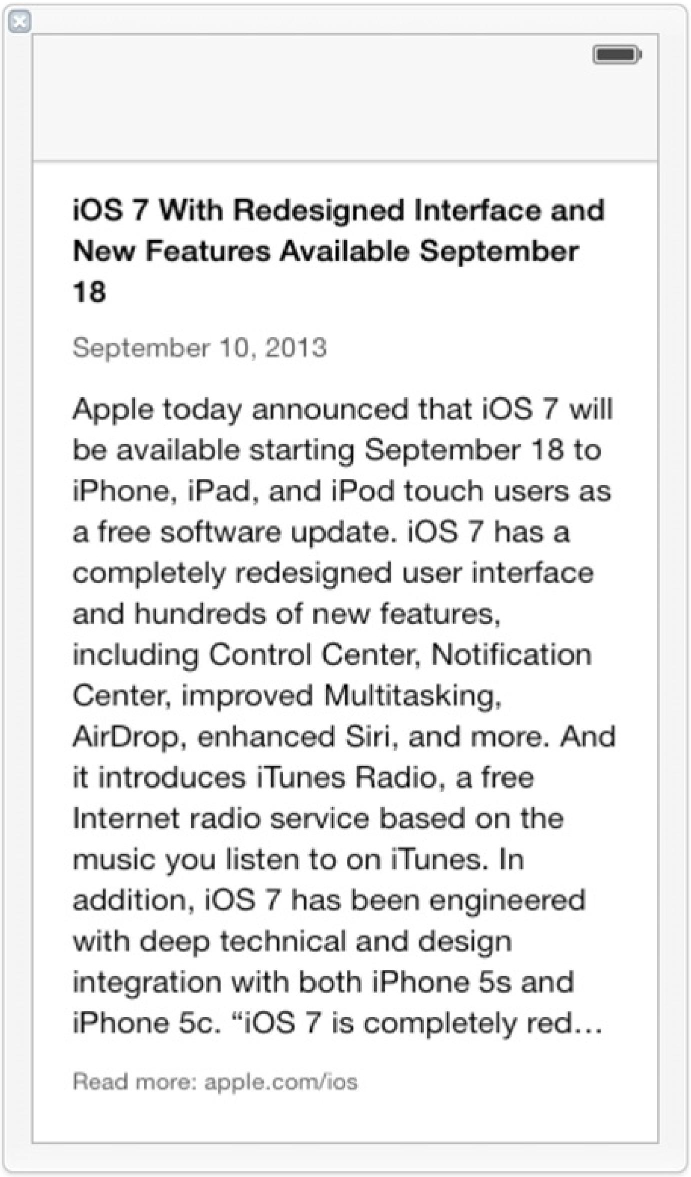
Size: Not Applicable

Autoshrink: Fixed Font Size

Tighten Letter Spacing

Repeat steps 1-3 for the subheadline label, the body label, and the footer label, choosing the appropriate text styles for each (Subhead, Body, and Footnote).

When it's all done, our view should look like:



iOS 7 With Redesigned Interface and New Features Available September 18

September 10, 2013

Apple today announced that iOS 7 will be available starting September 18 to iPhone, iPad, and iPod touch users as a free software update. iOS 7 has a completely redesigned user interface and hundreds of new features, including Control Center, Notification Center, improved Multitasking, AirDrop, enhanced Siri, and more. And it introduces iTunes Radio, a free Internet radio service based on the music you listen to on iTunes. In addition, iOS 7 has been engineered with deep technical and design integration with both iPhone 5s and iPhone 5c. “iOS 7 is completely red…

Read more: apple.com/ios

Unfortunately, configuring our view to support dynamic type isn't *quite* that simple. We'll actually need to write some code to size the text appropriately whenever the user changes the text size in the Settings app. To do that, we need to listen for a notification for the change in the preferred font size. The notification is called UIContentSizeCategoryDidChangeNotification, and we'll add our listener in our viewDidLoad:

```
[[NSNotificationCenter defaultCenter]
addObserver:self
selector:@selector(handleContentSizeDidChange:)
name:UIContentSizeCategoryDidChangeNotification
object:nil];
```

Here, we're just adding an observer that's listening for the change in text size - the UIContentSizeCategoryDidChangeNotification. And when we do observe that notification, we're going to call -[self handleContentSizeDidChange:] which will actually change the text style on our labels.

So now that we're *listening* for changes in text size, we need to actually *respond* to those changes when we receive the notification. When the user chooses a new font size, we can use `+[UIFont preferredFontForTextStyle:]` to get a font that's sized to the user's preference. You'll pass in the text style you want, choosing from the text styles listed previously. In our case, we're going to pass in either the Headline, Subheadline, Body, or Footer, depending on which label we're getting a font for. Below is our method that responds to the change in text size. It's important to note that this code *is* necessary - iOS does not do this automatically when you set the font to one of the Text Styles in Interface Builder. Here's that method:

```
- (void) handleContentSizeDidChange:(NSNotification*)aNotification
```

```
{
 self.headlineLabel.font = [UIFont preferredFontForTextStyle:UIFontTextStyleHeadline];
 self.subheadLabel.font = [UIFont preferredFontForTextStyle:UIFontTextStyleSubheadline];
 self.bodyLabel.font = [UIFont preferredFontForTextStyle:UIFontTextStyleBody];
 self.footnoteLabel.font = [UIFont preferredFontForTextStyle:UIFontTextStyleFootnote];
}
```

First we set the font for the headline label to the user's preferred font for the Headline text style. Then we set the font on the subhead label to the user's preferred font for Subheadline text style. Next we set the body label's font to the user's preferred font for the Body text style. And finally, we set the footnote label's font to the user's preferred font for the Footnote text style.

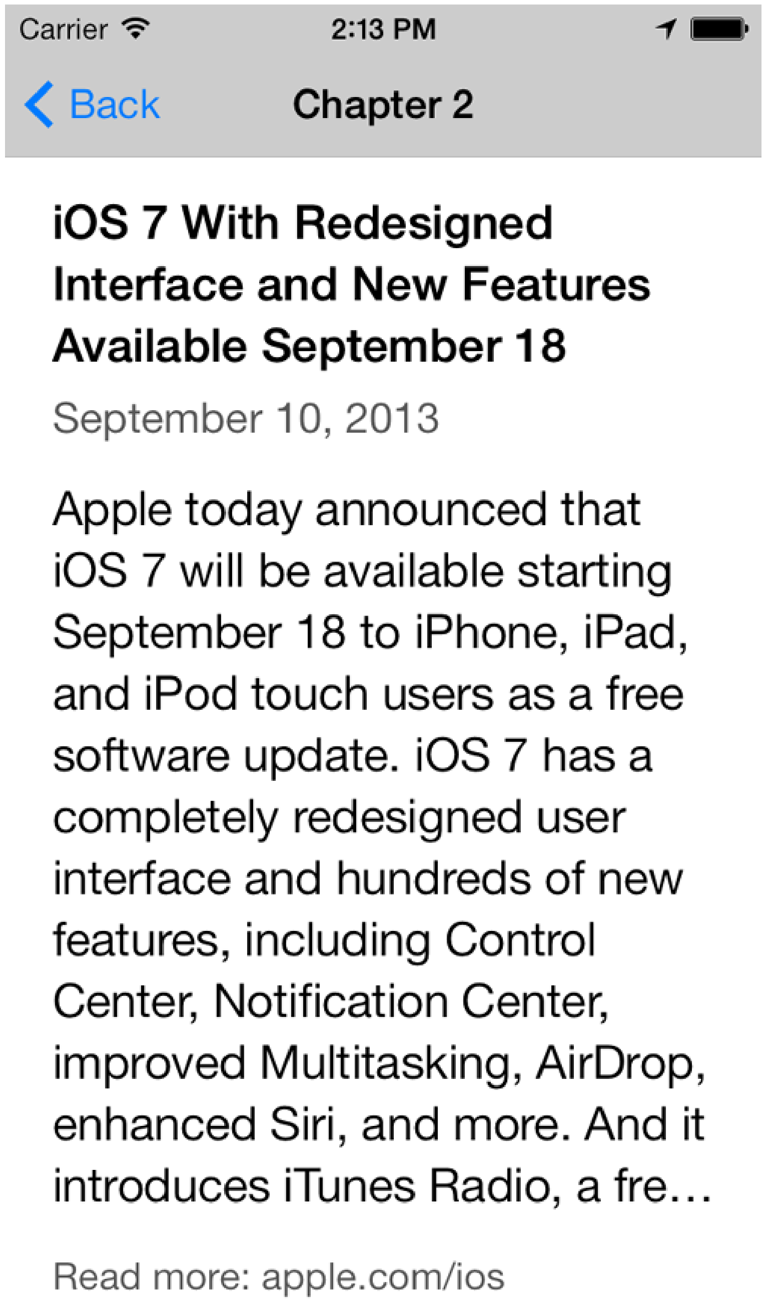
Now let's test this. Run the app and navigate to your Dynamic Type view. When you get there, make sure everything seems to be appropriately sized.

Changing the Dynamic Type text size

Now we need to change the text size to see our view update in response to a text size change. To set the text size:

1. Launch the Settings app
2. Select General > Text Size
3. Drag the slider to adjust the text size

First let's test it by sliding the size all the way up. Do that using the steps above, then go back to your app. Does your view look something like the following?



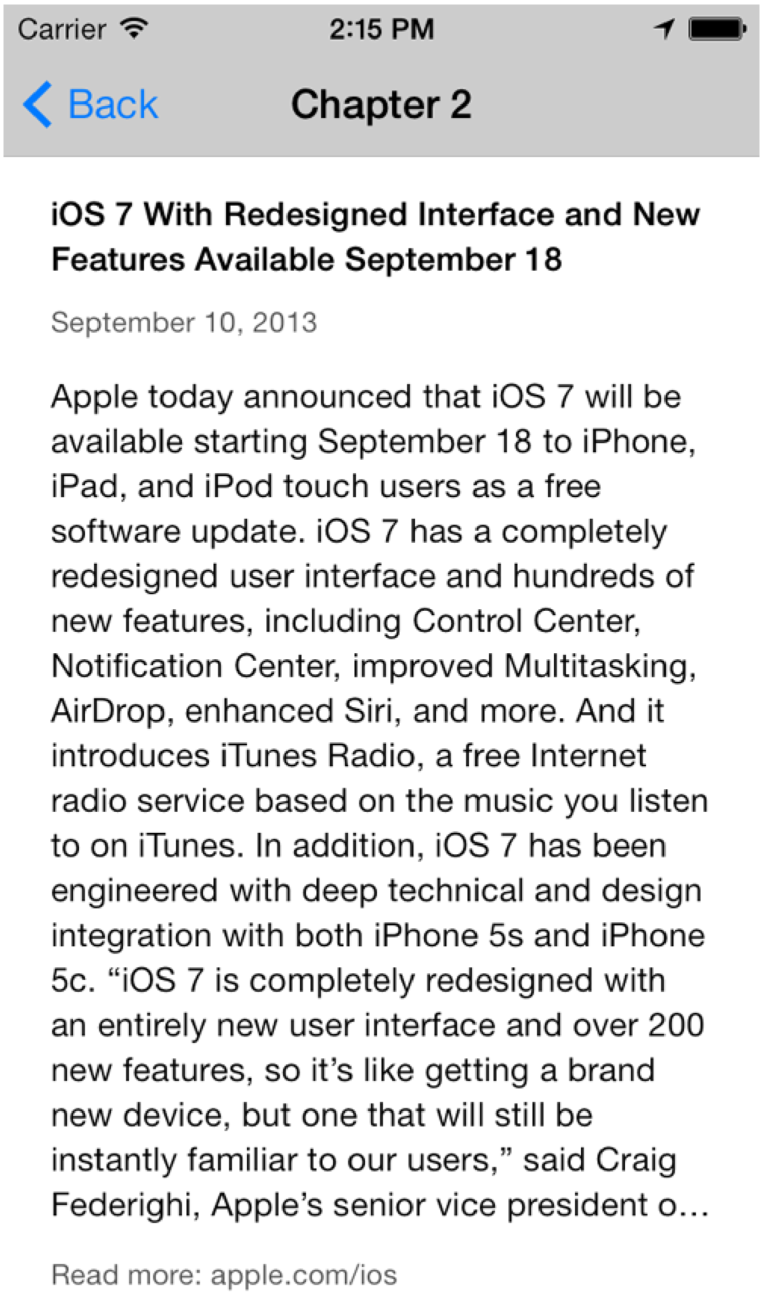
iOS 7 With Redesigned Interface and New Features Available September 18

September 10, 2013

Apple today announced that iOS 7 will be available starting September 18 to iPhone, iPad, and iPod touch users as a free software update. iOS 7 has a completely redesigned user interface and hundreds of new features, including Control Center, Notification Center, improved Multitasking, AirDrop, enhanced Siri, and more. And it introduces iTunes Radio, a fre…

Read more: apple.com/ios

Now let's go back to the Settings app and make the text size as small as possible. Go do that, again following the steps above, then go back to your view. It should look about like this:


Carrier 2:15 PM

Back Chapter 2

iOS 7 With Redesigned Interface and New Features Available September 18

September 10, 2013

Apple today announced that iOS 7 will be available starting September 18 to iPhone, iPad, and iPod touch users as a free software update. iOS 7 has a completely redesigned user interface and hundreds of new features, including Control Center, Notification Center, improved Multitasking, AirDrop, enhanced Siri, and more. And it introduces iTunes Radio, a free Internet radio service based on the music you listen to on iTunes. In addition, iOS 7 has been engineered with deep technical and design integration with both iPhone 5s and iPhone 5c. “iOS 7 is completely redesigned with an entirely new user interface and over 200 new features, so it’s like getting a brand new device, but one that will still be instantly familiar to our users,” said Craig Federighi, Apple’s senior vice president o…

Read more: apple.com/ios

Note that the user can also choose to make the font bold in the Accessibility Settings (Settings app > Accessibility > Bold Text) so be sure to test this as well.

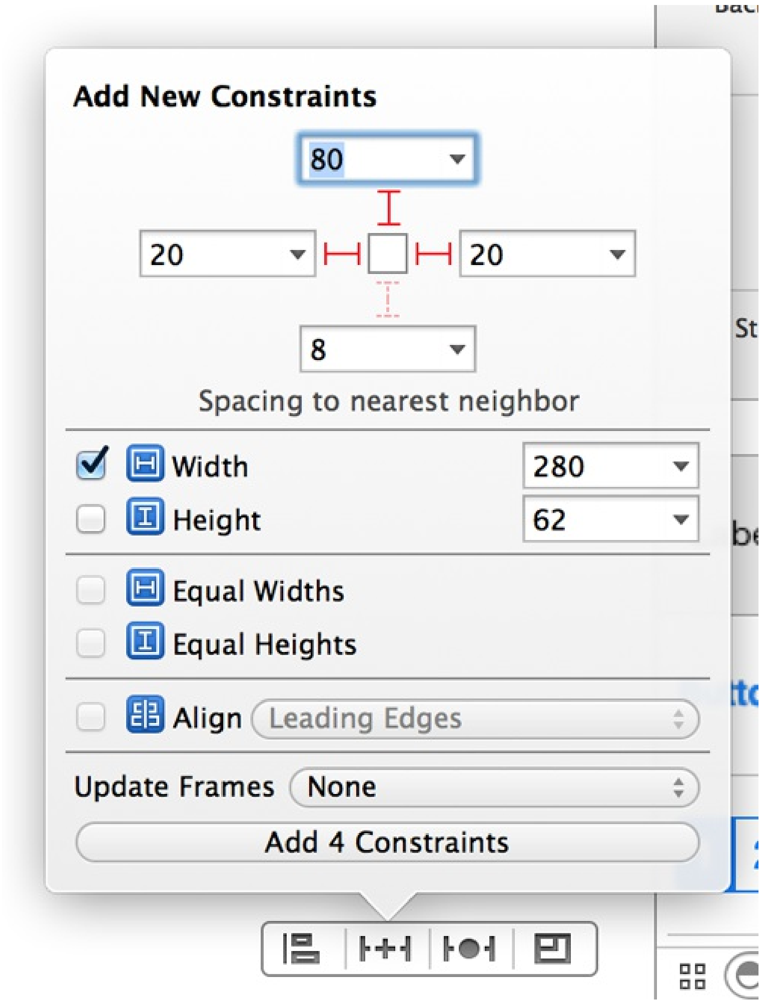
Autolayout in Xcode 5

Having our view dynamically size the text is great, but without laying out the views in response to changes in text size, our app isn't going to look very nice when the user resizes the text. Let's fix that problem now.

You may have noticed in your app that the views were positioned strangely after changes in text size. Autolayout can fix that for us. Let's look at our view in Interface Builder.

Select the headline label in Interface Builder, then in the bottom right of the canvas, choose the second icon from the left, which should look like a plus sign between bars, like this: http://cl.ly/image/2N262V3x043e

Configure the Autolayout constraints in this view by clicking the red bars to set the top, left, and right constraints. We also want to constrain the width to 280. When it's all said and done, it should look like this:


Add New Constraints

Top: 80

Left: 20

Right: 20

Bottom: 8

Spacing to nearest neighbor

[✓] Width 280

[ ] Height 62

[ ] Equal Widths

[ ] Equal Heights

[ ] Align Leading Edges

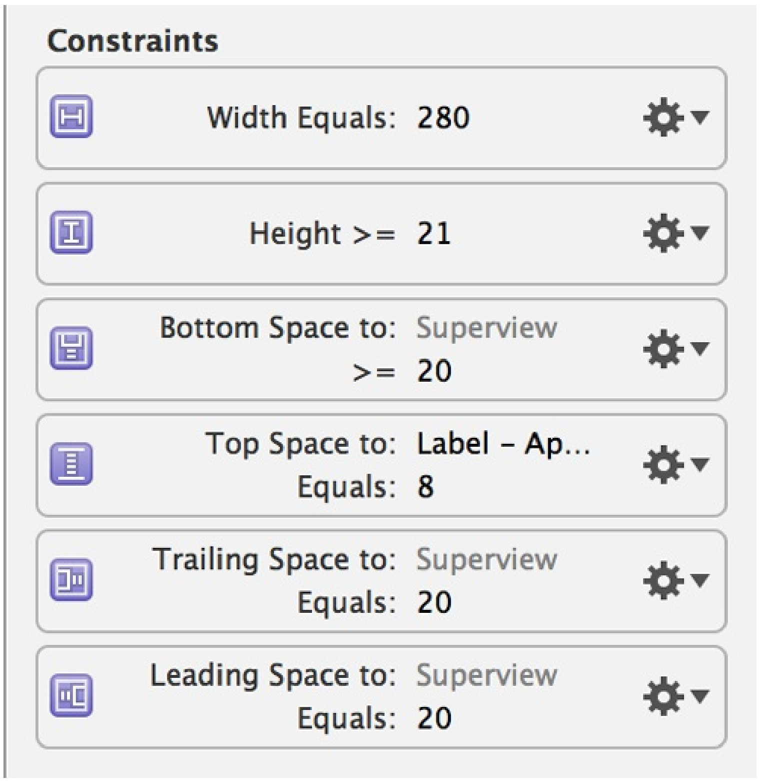
Update Frames None

Add 4 Constraints

Click the **Add 4 Constraints** button to add the constraints.

We'll want to similarly constrain the other labels. We want them relative to each other, though, rather than the top of the view. Xcode is pretty smart about this, however, so clicking the red bars for top, right and left as well as checking the Width box on the subheadline and body labels should do the trick.

For the footer label, we want to make sure it stays on the screen. We don't want the other views to push it off as they increase their height to fit a larger text size. And we'll let the body be truncated if it's too big to fit in the view, in favor of keeping the Read More label on the screen. We can do that by setting its height >= 21 and setting its bottom space constraint to be >= 20 pixels above the superview. And it's top space should be 8 pixels below the body. Below is a screen shot of what all of its constraints should look like. You can find these in the Utilities pane on the right, in the Size inspector.


Constraints

Width Equals: 280

Height >= 21

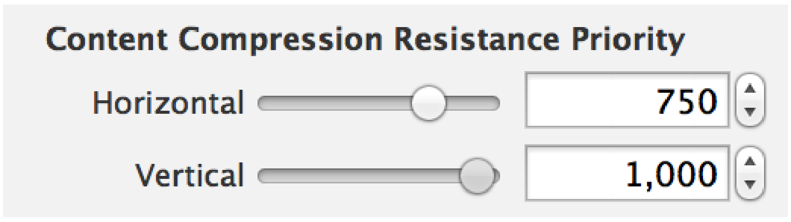
Bottom Space to: Superview >= 20

Top Space to: Label - Ap... Equals: 8

Trailing Space to: Superview Equals: 20

Leading Space to: Superview Equals: 20

You may have also noticed that the headline view truncates its text when the text size is large. We don't want this - we want to be able to see all the text in the headline - we can live with the body label being squished and the body text being truncated, but the headline should be visible. The way to do that is by setting the compression resistance on the headline label. With the headline label selected, look in the Size Inspector for the Content Compression Resistance Priority. Here, we want to set the Vertical to 1,000, telling iOS that we never want this label to be compressed vertically. Setting the priority to 1,000 makes it Required, so we can be sure that it won't be compressed. Here's how that looks:


Content Compression Resistance Priority

Horizontal 750

Vertical 1,000

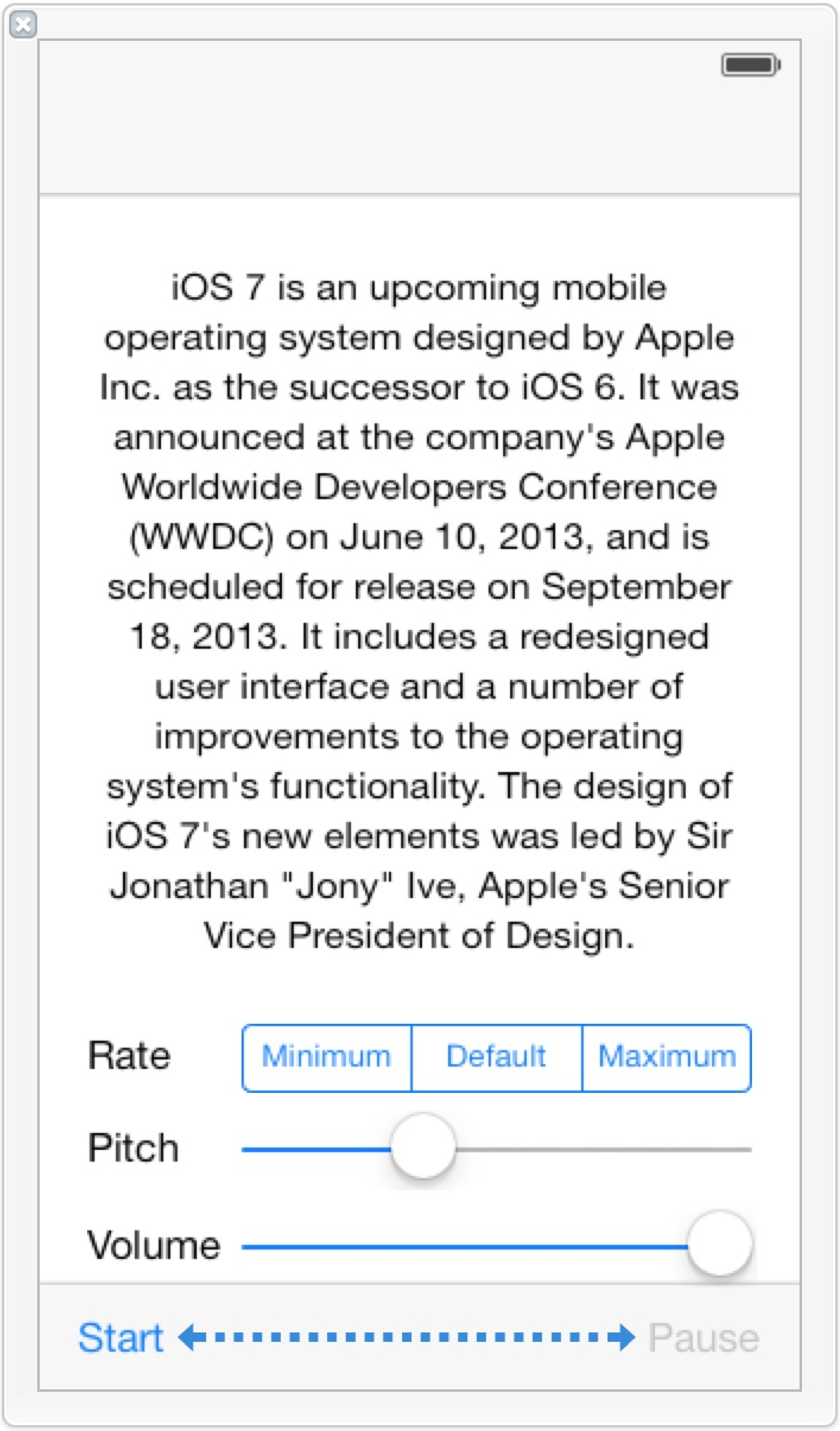
Text to Speech

Many apps already use text to speech, and until now, they've had to roll their own synthesizer and include a large bundle in the app. In iOS 7, Apple has introduced a Text to Speech API, so developers can quickly and easily take advantage of Text to Speech in their apps with only a few lines of code. It's really simple. Let's have it say hello to you, so put in your name in place of Troy:

```
AVSpeechSynthesizer *synthesizer = [[AVSpeechSynthesizer alloc] init];
AVSpeechUtterance *utterance = [AVSpeechUtterance speechUtteranceWithString:@"Hello, Troy!"];
[synthesizer speakUtterance:utterance];
```

To get this to run, you'll need to add AVFoundation.framework to your project and #import <AVFoundation/AVFoundation.h> in your implementation file.

The three lines above are enough to have iOS say hello to you, but there's so much more you can do. The API also allows you to configure the voice, volume, pitch, rate, and more. You can also start, stop, pause, and resume speech with the APIs. This makes it trivial to add a few buttons that control these and have iOS speak to you. Let's do that now. Here's what we're shooting for:


iOS 7 is an upcoming mobile operating system designed by Apple Inc. as the successor to iOS 6. It was announced at the company's Apple Worldwide Developers Conference (WWDC) on June 10, 2013, and is scheduled for release on September 18, 2013. It includes a redesigned user interface and a number of improvements to the operating system's functionality. The design of iOS 7's new elements was led by Sir Jonathan "Jony" Ive, Apple's Senior Vice President of Design.

You can use any text you want for the label (or you could do this entirely without a label), but we've chosen our sample from Wikipedia's iOS 7 page.

Configuring the speech rate, pitch, and volume

The Rate segmented control sets the rate at which iOS reads the text. The three values pictured are provided by Apple as constants, but the rate is actually just a float whose value is between AVSpeechUtteranceMinimumSpeechRate (or 0) and AVSpeechUtteranceMaximumSpeechRate (or 1), meaning you could set your own value. The default (AVSpeechUtteranceDefaultSpeechRate) is set to 0.5. Setting the rate is simple:

```
utterance.rate = AVSpeechUtteranceMaximumSpeechRate;
```

With the code above, we're just setting the rate to the maximum. But since we have a control for setting the rate, our code is a bit more complex:

```
CGFloat rate = AVSpeechUtteranceDefaultSpeechRate;
switch (self.rateControl.selectedSegmentIndex)
{
case 0:
rate = AVSpeechUtteranceMinimumSpeechRate;
break;
case 1:
rate = AVSpeechUtteranceDefaultSpeechRate;
break;
case 2:
rate = AVSpeechUtteranceMaximumSpeechRate;
break;
default:
break;
}
utterance.rate = rate;
```

In this case, we're just switching on the selected segment of the segmented control and setting the rate to the corresponding value. No surprises here.

The Pitch slider controls the pitch multiplier of the voice, which controls how high or low the pitch of the voice is. Values for this range from 0.5 for the lowest pitch to 2.0 for the highest. The default pitch is 1.0. We've configured our slider to only allow these values, and set its default to 1.0. Here's our code that sets the pitch:

```
utterance.pitchMultiplier = self.pitchSlider.value;
```

The Volume slider controls the volume of the voice. Values range from 0 to 1 for this, where 0.0 is silent and 1.0 is the loudest possible. The default value here is 1.0, so we've configured our slider accordingly. Below is the code that sets the volume:

```
utterance.volume = self.volumeSlider.value;
```

The voice can also be configured on the utterance, so if you'd like to change the voice for a different language or region, that's an option as well. For English, there are several options:

```
en-GB
en-US
en-AU
en-IE
en-ZA
```

Here's an example of how to set the voice:

```
utterance.voice = [AVSpeechSynthesisVoice voiceWithLanguage:@"en-GB"];
```

We've chosen to have the voice talk to us in British English, but you're free to choose any language and region you like. Since our text is in English, we'd recommend trying one of the English voices. :) Note that if you don't set the voice on the utterance object, it'll use the default based upon your region and language settings, so there's no need to configure this property.

The rate, pitch, volume, and voice are properties on the AVSpeechUtterance object and can only be configured before asking the synthesizer to speak the utterance. And it's worth pointing out that you can't change the AVSpeechUtterance's string - it's readonly.

Starting, stopping, pausing, and resuming speech

Let's now examine how to stop, start, pause and resume speech in our app.

Starting and stopping speech synthesis

Now let's look at how to start and stop speech with the AVSpeechSynthesizer. It's quite simple, really, thanks to Apple's wonderful APIs. Here are the methods we'll use:

```
- (void)speakUtterance:(AVSpeechUtterance *)utterance
- (BOOL)stopSpeakingAtBoundary:(AVSpeechBoundary)boundary
```

Yep, it's really that simple. Let's look at the first - starting the speech synthesizer. Here's how to do it:

```
AVSpeechUtterance *utterance = [AVSpeechUtterance speechUtteranceWithString:@"Hello, there!"];
 [synthesizer speakUtterance:utterance];
```

As we've seen before, we just need to create an utterance with a string and ask the synthesizer to speak it. Stopping is just as easy:

```
[synthesizer stopSpeakingAtBoundary:AVSpeechBoundaryImmediate];
```

In the above line, we're asking the synthesizer to stop speaking immediately, canceling the current utterance and removing any others from the synthesizer's queue* (see The Synthesizer's Queue). Another option for the boundary parameter is AVSpeechBoundaryWord, which asks the synthesizer to stop speaking after the word that's currently being spoken.

Aside: The Synthesizer's Queue
The AVSpeechSynthesizer has a queue, so whenever we call its `-speakUtterance:` method, the utterance we pass to it is added to the synthesizer's queue. If it's the only utterance in the queue, the synthesizer starts speaking it immediately. If there are other utterances in the queue, the utterance we pass will be spoken when the other utterances finish and it gets to the beginning of the queue. So if we want the synthesizer to say hello and ask how we're doing, we could do it like this:

```
AVSpeechUtterance *helloUtterance = [AVSpeechUtterance speechUtteranceWithString:@"Hello, there!"];
 AVSpeechUtterance *howAreYouUtterance = [AVSpeechUtterance speechUtteranceWithString:@"How are you?"];
 [synthesizer speakUtterance:helloUtterance];
 [synthesizer speakUtterance:howAreYouUtterance];
```

It's important to note that the howAreYouUtterance gets queued - it doesn't speak over or cut off the helloUtterance. A use case for this might be an app that allows children to build sentences by arranging words (or images) in order. When they tap the "speak" button, you could take the words they've arranged and call `-speakUtterance:` with an utterance for each string, rather than having to concatenate each string into a single utterance.

Let's move on to pausing and resuming speech.

Pausing and resuming speech synthesis

Next up, we'll look at pausing and resuming speech on the AVSpeechSynthesizer. We'll need the following two methods.

```
- (BOOL)pauseSpeakingAtBoundary:(AVSpeechBoundary)boundary
- (BOOL)continueSpeaking
```

Let's start with the first - pausing speech. Here's a sample of how we can pause speech at the end of the word that's currently being spoken:

```
[synthesizer pauseSpeakingAtBoundary:AVSpeechBoundaryWord];
```

On our synthesizer instance, we ask it to pause speaking at the word boundary. The other option is to request for it to stop speaking immediately, passing `AVSpeechBoundaryImmediate` for the boundary.

Getting notified about speech starts, stops, pauses, and resumes

The AVSpeechSynthesizerDelegate protocol defines methods that allow the delegate of AVSpeechSynthesizer to be notified when the synthesizer pauses or resumes, starts or finishes a block of text, or as it produces each individual unit of speech. For our purposes, we'll look at pausing, resuming, starting, and finishing.

Let's start with the method that gets called when speech starts:

```
- (void)speechSynthesizer:(AVSpeechSynthesizer *)synthesizer didStartSpeechUtterance:(AVSpeechUtterance *)utterance
```

As you can see, we get a reference to the synthesizer and the utterance in this delegate method. In our case, we want to set up our view for the started state:

```
- (void)speechSynthesizer:(AVSpeechSynthesizer *)synthesizer didStartSpeechUtterance:(AVSpeechUtterance *)utterance
{
 [self setViewForState:BEPReadingStateStarted];
}
```

We've defined an enum that tells us which state we're in, and we're asking our setViewForState method to get our view in order for the "started" state. Let's take a look at that method.

```
- (void) setViewForState:(BEPReadingState)state
{
 switch (state)
 {
 case BEPReadingStateStarted:
 [self setStartStopButtonToStart:NO];
 [self setPauseResumeButtonToPause:YES];
 self.rateControl.enabled = NO;
 self.pitchSlider.enabled = NO;
 self.volumeSlider.enabled = NO;
 self.pauseResumeButton.enabled = YES;
 break;
 case BEPReadingStateStopped:
 [self setStartStopButtonToStart:YES];
 [self setPauseResumeButtonToPause:YES];
 self.rateControl.enabled = YES;
 self.pitchSlider.enabled = YES;
 self.volumeSlider.enabled = YES;
 self.pauseResumeButton.enabled = NO;
 break;
 case BEPReadingStatePaused:
 [self setPauseResumeButtonToPause:NO];
 break;
 case BEPReadingStateResumed:
 [self setPauseResumeButtonToPause:YES];
 break;
 default:
 break;
 }
}

```

So since we're in the "started" state here, let's focus just on that case. Here's the meaningful code from it:

```
[self setStartStopButtonToStart:NO];
 [self setPauseResumeButtonToPause:YES];
 self.rateControl.enabled = NO;
 self.pitchSlider.enabled = NO;
 self.volumeSlider.enabled = NO;
 self.pauseResumeButton.enabled = YES;
```

First, we want to set the start/stop button to the proper state, and since the state is Started, we want that button to say "Stop". (We'll look at the `-setStartStopButtonToStart:` method a bit later). The next line asks our view controller to set the pause/resume to "Pause". (Again, we'll look at this method later). In the next three lines, we disable the controls for rate, pitch, and volume. Why, you ask? An excellent question, dear reader. Once speech has started, it's impossible to change the rate, pitch, or volume of the voice, so we're showing that in our UI. Finally, we want to enable the pause/resume button, as speech is started now and the user should be able to pause it.

We'll skip over the rest of the cases in the `-setViewForState:` method, as we trust you're able to figure them out based on the breakdown of the first case above. So now let's take a look at that `-setStartStopButtonToStart:` method we glossed over before.

```
- (void) setStartStopButtonToStart:(BOOL)start
{
 if (start)
 {
 self.startStopButton.title = NSLocalizedString(@"Start", nil);
 [self.startStopButton setAction:@selector(startSpeaking:)];
 }
 else
 {
 self.startStopButton.title = NSLocalizedString(@"Stop", nil);
 [self.startStopButton setAction:@selector(stopSpeaking:)];
 }
}

```

This is relatively simple, straightforward code that toggles the start/stop button between the start and stop states. If start is YES, we enter the first block of the if statement, setting the title of the button to "Start" and setting the action to `startSpeaking:`. In the else block, we do quite the opposite - the button's title becomes "Stop" and the action for the button becomes `stopSpeaking:`. Onward and upward - let's see those methods.

```
- (IBAction) startSpeaking:(id)sender<
{
 AVSpeechUtterance *utterance = [AVSpeechUtterance speechUtteranceWithString:self.bodyLabel.text];

 utterance.voice = [AVSpeechSynthesisVoice voiceWithLanguage:@"en-GB"];
 utterance.volume = self.volumeSlider.value;<br /> utterance.pitchMultiplier = self.pitchSlider.value;

 CGFloat rate = AVSpeechUtteranceDefaultSpeechRate;
 switch (self.rateControl.selectedSegmentIndex)
 {
case 0:
 rate = AVSpeechUtteranceMinimumSpeechRate;
 break;
 case 1:
 rate = AVSpeechUtteranceDefaultSpeechRate;
 break;
 case 2:
 rate = AVSpeechUtteranceMaximumSpeechRate;
 break;
 default:
 break;
 }
 utterance.rate = rate;

 [self.synthesizer speakUtterance:utterance];
}
```

Here we have a bit more code, but again it's relatively straightforward. Let's break it down.

```
AVSpeechUtterance *utterance = [AVSpeechUtterance speechUtteranceWithString:self.bodyLabel.text];
```

Just create a speech utterance with the bodyLabel's text. Later we'll ask the synthesizer to speak this utterance, but first, let's configure it! How about a British English voice?

utterance.voice = [AVSpeechSynthesisVoice voiceWithLanguage:@"en-GB"];

Creating a voice with a language is quite simple, as you can see - all we need to do is pass it a proper language and localization. To see all of the possible voices, you can call `+[AVSpeechSynthesisVoice speechVoices]` Moving on, let's set the volume of the utterance.

```
utterance.volume = self.volumeSlider.value;
```

The volume property on the utterance sets - you guessed it - the volume of the speech. We get the value for the volume from the volumeSlider in the UI, which we've configured in Interface Builder with a minimum value of 0.0 and a maximum of 1.0. And yep, you guessed it, that corresponds with the range of possible values for the utterance's volume. If we don't set the volume, the utterance uses the default value of 1.0. Next up: pitch!

utterance.pitchMultiplier = self.pitchSlider.value;

The pitch multiplier on the utterance sets the pitch of the speech. Again we're getting the value from a slider in the UI. We made sure to configure the pitchSlider in Interface Builder to its minimum and maximum values: 0.5 and 2.0. The pitchMultiplier on the utterance defaults to 1.0, so we've also configured this in Interface Builder. Now we'll look at the rate.

```
CGFloat rate = AVSpeechUtteranceDefaultSpeechRate;
 switch (self.rateControl.selectedSegmentIndex)<
 {
 case 0:
 rate = AVSpeechUtteranceMinimumSpeechRate;
 break;
 case 1:
 rate = AVSpeechUtteranceDefaultSpeechRate;
 break;
 case 2:
 rate = AVSpeechUtteranceMaximumSpeechRate;
 break;
 default:
 break;
 }
 utterance.rate = rate;
```

With the switch statement, the code to set the rate looks a bit more complex, but in reality, it's simple. The rate on the utterance is a float and can be a value between 0.0 and 1.0, but Apple provides us with a few constants: AVSpeechUtteranceMinimumSpeechRate, AVSpeechUtteranceDefaultSpeechRate, and AVSpeechUtteranceMaximumSpeechRate. We easily could have made this a slider like the pitch and volume with a range from 0.0 to 1.0, but we opted to illustrate the constants Apple provides. Feel free to set your rate to crazy non-constant values like 0.7 or even 0.125 and see how it works! :)

That brings us to the code that finally starts speech in this method:

```
[self.synthesizer speakUtterance:utterance];
```

Now that we've configured our utterance, we just pass it to the synthesizer and ask it to speak. This will cause the synthesizer to start speaking immediately.

Stop delegate method

When the synthesizer delegate notifies us that speech has either finished or been canceled, we only need to set up the view for the stopped state. Here's what the finish method looks like:

```
- (void)speechSynthesizer:(AVSpeechSynthesizer *)synthesizer didFinishSpeechUtterance:(AVSpeechUtterance *)utterance
{
 [self setViewForState:BEPReadingStateStopped];
}
```

We just call the setViewForState, telling it that the state is now "Stopped." And we've seen the `-setViewForState:` method before, but let's look at the stopped case again:

```
case BEPReadingStateStopped:
 [self setStartStopButtonToStart:YES];
 [self setPauseResumeButtonToPause:YES];
 self.rateControl.enabled = YES;
 self.pitchSlider.enabled = YES;
 self.volumeSlider.enabled = YES;
 self.pauseResumeButton.enabled = NO;
 break;
```

First we change the start/stop button to "Start" with the call to `-setStartStopButtonToStart`. Then we set the pause/resume button to "Pause" so it's ready for the next time speech is started. We the enable all of the controls that configure the utterance - the rateControl, pitchSlider, and volumeSlider. Since speech is now stopped, we can again allow the user to configure those settings. And finally, we disable the pause/resume button since we can't pause or resume speech when it's stopped. Next up, we'll look at the pause delegate method.

Pause delegate method

In the delegate method that notifies us that speech was paused, we'll ask our view controller to configure the view for our paused state. Here's that delegate method:

```
- (void)speechSynthesizer:(AVSpeechSynthesizer *)synthesizer didPauseSpeechUtterance:(AVSpeechUtterance *)utterance
{
 [self setViewForState:BEPReadingStatePaused];
}
```

The one line in this method again calls our `-setViewForState:` method, passing in the Paused state. That brings us to the Paused case in `-setViewForState:`:

case BEPReadingStatePaused:
[self setPauseResumeButtonToPause:NO];
break;

And in the `-setPauseResumeButtonToPause:`, we simply set the title and action on our pause/resume button.

```
- (void) setPauseResumeButtonToPause:(BOOL)paused
{
 if (paused)
 {
 self.pauseResumeButton.title = NSLocalizedString(@"Pause", nil);
 [self.pauseResumeButton setAction:@selector(pauseSpeaking:)];
 }
 else
 {
 self.pauseResumeButton.title = NSLocalizedString(@"Resume", nil);
[self.pauseResumeButton setAction:@selector(resumeSpeaking:)];
 }
}
```

Since we've called this method with NO for the `paused` parameter, let's look at the `else` block. In it, we first set the title of the pause/resume button to "Resume". Then we set the action to resumeSpeaking, so when the user taps our "Resume" button, we'll do this:

```
- (void) resumeSpeaking:(id)sender
{
 [self.synthesizer continueSpeaking];
}
```

This method calls the continueSpeaking on our AVSpeechSynthesizer instance, asking it to continue speaking from where it left off when we paused it. That takes us to our continue delegate method, which is what gets called as soon as `-continueSpeaking:` is called in the method above. Let's look at that delegate method:

```
- (void)speechSynthesizer:(AVSpeechSynthesizer *)synthesizer didContinueSpeechUtterance:(AVSpeechUtterance *)utterance
{
 [self setViewForState:BEPReadingStateResumed];
}
```

And all we do is a single call our `-setViewForState:` method, passing the Resumed state in. Let's look at the case in that method where we handle the Resumed state:

case BEPReadingStateResumed:
[self setPauseResumeButtonToPause:YES];
break;

This asks our view controller to set the pause/resume button to its pause state, so let's look at that now:

```
- (void) setPauseResumeButtonToPause:(BOOL)paused
{
 if (paused)
 {
 self.pauseResumeButton.title = NSLocalizedString(@"Pause", nil);
 [self.pauseResumeButton setAction:@selector(pauseSpeaking:)];
 }
 else
 {
 self.pauseResumeButton.title = NSLocalizedString(@"Resume", nil);
 [self.pauseResumeButton setAction:@selector(resumeSpeaking:)];
 }
}
```

Here we've passed YES into the method for the paused parameter, so let's break down the `if` block. First, we'll set the title of the pause/resume button to "Pause". Then we'll set the selector so it calls `-pauseSpeaking:` when the button is tapped. So let's assume the user has tapped the button and look at what happens next.

```
- (IBAction) pauseSpeaking:(id)sender
{
 [self.synthesizer pauseSpeakingAtBoundary:AVSpeechBoundaryImmediate];
}
```

And we're finally back to the synthesizer. We send a message to the synthesizer asking it to pause speaking immediately. You may remember from before that we can ask it to pause speaking after the current word by passing in AVSpeechBoundaryWord instead.

And that's that - now you can go build apps that talk to your users with AVFoundation's simple, easy-to-use API.

Summary

In this chapter, we looked at why you should make your apps accessible (in case you forgot - it's easy, it's right, and it's good for everyone). We also explored Dynamic Type and how your apps can respond to the user's preferred text size. And we wrapped it all up with Apple's new Text to Speech API and showed how you can implement Text to Speech in your apps.