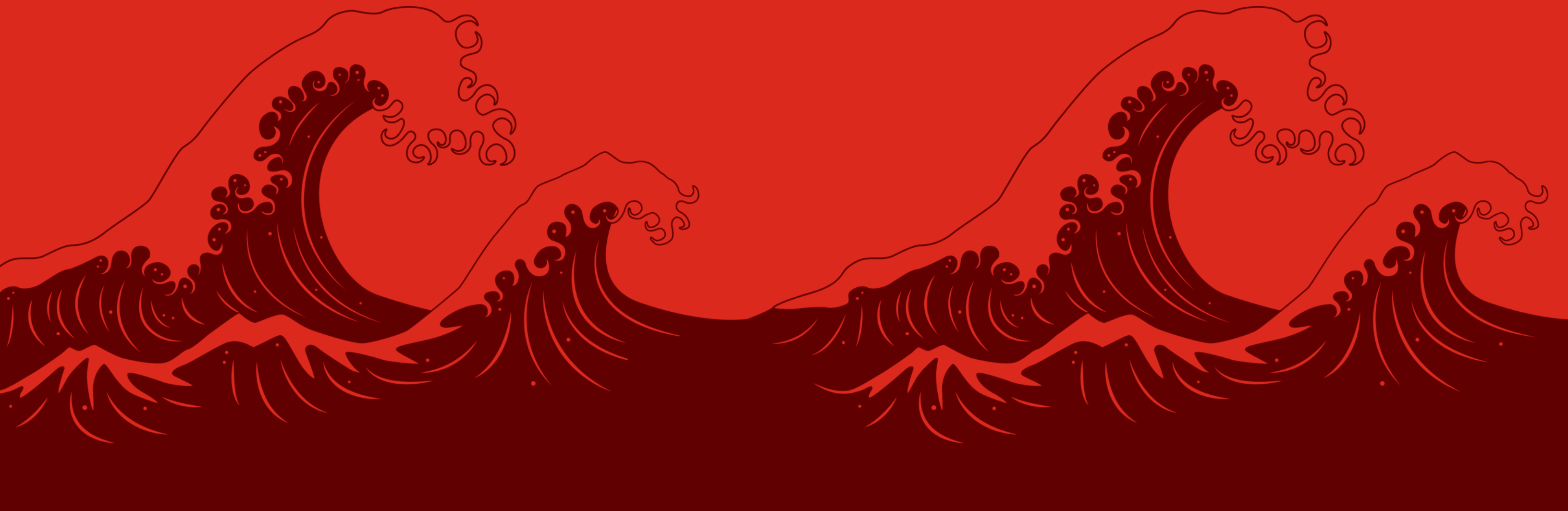
iOS 7 day by day

a review of iOS 7 for developers, in 24 bite-sized chunks



shinobicontrols

by Sam Davies

iOS7 Day by Day

a review of iOS7 for developers, in 24 bite-sized chunks

Sam Davies

This book is for sale at http://leanpub.com/ios7daybyday

This version was published on 2013-11-05

Tweet This Book!

Please help Sam Davies by spreading the word about this book on Twitter!

The suggested hashtag for this book is #iOS7DayByDay.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#iOS7DayByDay

Contents

Preface

Welcome along to iOS7 Day-by-Day! In September of 2013 Apple released the 7th version of their exceedingly popular mobile operating system into the world. With it came a new user interface appearance, new icons and lots of other little changes for users to complain about. However, the most exciting changes were, as ever, in the underlying APIs - with new frameworks and considerable new functionality added to existing frameworks.

There are so many changes in fact that it's very difficult for a busy developer to pore through the release notes to discover the features which they can take advantage of. Therefore I wrote and published a daily blog series[1], in which each article discussed a new feature, and created a sample app to demonstrate it.

This series was very successful, and ran for a total of 24 days - covering many parts of the new operating system, including both the big headline frameworks and also the somewhat smaller hidden gems. The only notable omissions are the game-related frameworks, such as SpriteKit and changes to GameCenter. This is, unapologetically, because I have little experience of games, and also felt that these were being covered extensively elsewhere.

This book represents the sum-total of the blog series - each chapter represents a different post in the day-by-day series, with only minor changes. The original posts are still available online, and may offer some additional information in the form of comments.

If you have any comments or corrections for the book then do let me know - I'm @iwantmyrealname[2] on twitter.

Audience

Each chapter in this book is about a feature which was introduced in iOS7, and therefore is primarily targeted at developers who have had some experience of building iOS apps. Having said that, non-developers familiar with iOS might be interested in reading the new features available.

If you are new to iOS development it's probably worth reading through some of the introductory material available elsewhere - e.g. the excellent tutorials available on raywenderlich.com[3].

Book layout

This book is a collection of daily blog posts, which on the most-part stand alone. There are one or two which cross-reference each other, but they can be read entirely independently.

The chapters aren't meant to be complete tutorials, and as such, the code snippets within each chapter usually just highlight the more salient bits of code associated with a particular step. However, each chapter has an accompanying working app, the source code for which can be found on GitHub.

[1]http://www.shinobicontrols.com/blog/posts/2013/09/19/introducing-ios7-day-by-day/
[2]https://twitter.com/iwantmyrealname
[3]http://www.raywenderlich.com

Source code

The GitHub repository at github.com/ShinobiControls/ios7-day-by-day[4] contains projects which accompany each chapter, organized by day number.

The projects are all built using Xcode 5, and should run straight after downloading. Any pull-requests for fixes and improvements will be greatly appreciated!

[4]https://github.com/ShinobiControls/ios7-day-by-day

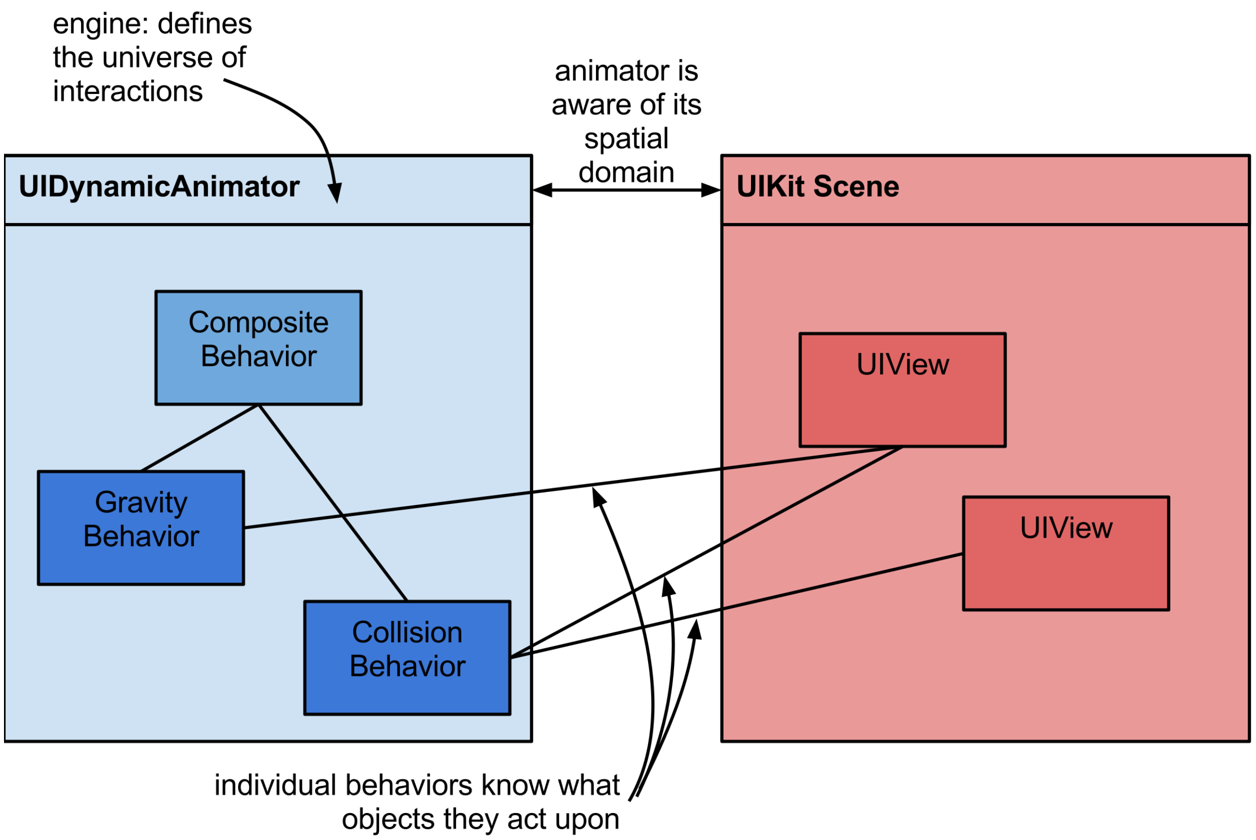
Day 0: UIKit Dynamics

With the introduction of iOS7 Apple made it very clear that they are pushing the interaction between devices and the real world. One of the new APIs they introduced was UIKit Dynamics - a 2-dimensional physics engine which lies underneath the entirety of UIKit. In day 0 of this blog series we're going to take a look at UIKit Dynamics and build a Newton's cradle simulation.

The physical universe

In order to model the physics of real world we use `UIDynamicBehavior` subclasses to apply different behaviors to objects which adopt the `UIDynamicItem` protocol. Examples of behaviors include concepts such as gravity, collisions and springs. Although you can create your own objects which adopt the `UIDynamicItem` protocol, importantly `UIView` already does this. The `UIDynamicBehavior` objects can be composited together to generate a behavior object which contains all the behavior for a given object or set of objects.

Once we have specified the behaviors for our dynamic objects we can provide them to a `UIDynamicAnimator` instance - the physics engine itself. This runs the calculations to determine how the different objects should interact given their behaviors. The follows shows a conceptual overview of the UIKit Dynamics world:


engine: defines the universe of interactions
animator is aware of its spatial domain
UIDynamicAnimator
UIKit Scene
Composite Behavior
Gravity Behavior
Collision Behavior
UIView
UIView
individual behaviors know what objects they act upon

UIKit Dynamics Conceptual Overview

Building a pendulum

Remembering back to high school science - one of the simplest objects studied in Newtonian physics is a pendulum. Let's create a `UIView` to represent the ball-bearing:

```
UIView *ballBearing = [[UIView alloc] initWithFrame:CGRectMake(0,0,40,40)];
ballBearing.backgroundColor = [UIColor lightGrayColor];
ballBearing.layer.cornerRadius = 10;
ballBearing.layer.borderColor = [UIColor grayColor].CGColor;
ballBearing.layer.borderWidth = 2;
ballBearing.center = CGPointMake(200, 300);
[self.view addSubview:ballBearing];
```

Now we can add some behaviors to this ball bearing. We'll create a composite behavior to collect the behavior together:

```
UIDynamicBehavior *behavior = [[UIDynamicBehavior alloc] init];
```

Next we'll start adding the behaviors we wish to model - first up gravity:

```
UIGravityBehavior *gravity = [[UIGravityBehavior alloc] initWithItems:@[ballBearing]];
gravity.magnitude = 10;
[behavior addChildBehavior:gravity];
```

`UIGravityBehavior` represents the gravitational attraction between an object and the Earth. It has properties which allow you to configure the vector of the gravitational force (i.e. both magnitude and direction). Here we are increasing the magnitude of the force, but keeping it directed in an increasing `y` direction.

The other behavior we need to apply to our ball bearing is an attachment behavior - which represents the string from which it hangs:

```
CGPoint anchor = ballBearing.center;
anchor.y -= 200;
UIAttachmentBehavior *attachment = [[UIAttachmentBehavior alloc]
                                        initWithItem:ballBearing attachedToAnchor:anchor];
[behavior addChildBehavior:attachment];
```

`UIAttachmentBehavior` instances attach dynamic objects either to an anchor point or to another object. They have properties which control the behavior of the attaching string - specifying its frequent, damping and length. The default values for this ensure a completely rigid attachment, which is what we want for a pendulum.

Now the behaviors are specified on the ball bearing we can create the physics engine to look after it all, which is defined as an ivar `UIDynamicAnimator *_animator;`:

```
_animator = [[UIDynamicAnimator alloc] initWithReferenceView:self.view];
[_animator addBehavior:behavior];
```

`UIDynamicAnimator` represents the physics engine which is required to model the dynamic system. Here we create it and specify which view it should use as its reference view (i.e. specifying the spatial universe) and add the composite behavior we've built.

With that we've actually created our first UIKit Dynamics system. However, if you run up the app, nothing will happen. This is because the system starts in and equilibrium state - we need to perturb the system to see some motion.

Gesture responsive behaviors

We need to add a gesture recognizer to the ball bearing to allow the user to play with the pendulum:

```
UIPanGestureRecognizer *gesture = [[UIPanGestureRecognizer alloc] initWithTarget:self
                                                action:@selector(handleBallBearingPan:)];
[ballBearing addGestureRecognizer:gesture];
```

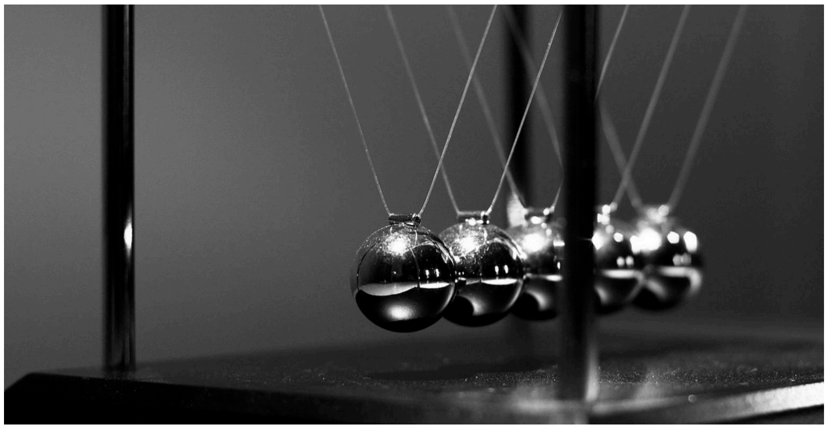
In the target for the gesture recognizer we apply a constant force behavior to the ball bearing:

```
- (void)handleBallBearingPan:(UIPanGestureRecognizer *)recognizer
{
    // If we're starting the gesture then create a drag force
    if (recognizer.state == UIGestureRecognizerStateBegan) {
        if(_userDragBehavior) {
            [_animator removeBehavior:_userDragBehavior];
        }
        _userDragBehavior = [[UIPushBehavior alloc] initWithItems:@[recognizer.view]
                                                               mode:UIPushBehaviorModeContinuous];
        [_animator addBehavior:_userDragBehavior];
    }

    // Set the force to be proportional to distance the gesture has moved
    _userDragBehavior.pushDirection =
                            CGVectorMake([recognizer translationInView:self].x / 10.f, 0);


    // If we're finishing then cancel the behavior to 'let-go' of the ball
    if (recognizer.state == UIGestureRecognizerStateEnded) {
        [_animator removeBehavior:_userDragBehavior];
        _userDragBehavior = nil;
    }
}
```

`UIPushBehavior` represents a simple linear force applied to objects. We use the callback to apply a force to the ball bearing, which displaces it. We have an ivar `UIPushBehavior *_userDragBehavior` which we create when a gesture start, remembering to add it to the dynamics animator. We set the size of the force to be proportional to the horizontal displacement. In order for the pendulum to swing we remove the push behavior when the gesture has ended.

Combining multiple pendulums

A Newton's cradle is an arrangement of identical pendulums, such that the ball bearings are almost touching.



Newton's Cradle

To recreate this using UIKit Dynamics we need to create multiple pendulums - following the same pattern for each of them as we did above. They should be spaced so that they aren't quite touching (see the sample code for details).

We also need to add a new behavior which will describe how the ball bearings collide with each other. We now have an ivar to store the ball bearings `NSArray *_ballBearings;`:

```
UICollisionBehavior *collision = [[UICollisionBehavior alloc]
                                              initWithObjects:_ballBearings];
[behavior addChildBehavior:collision];
```

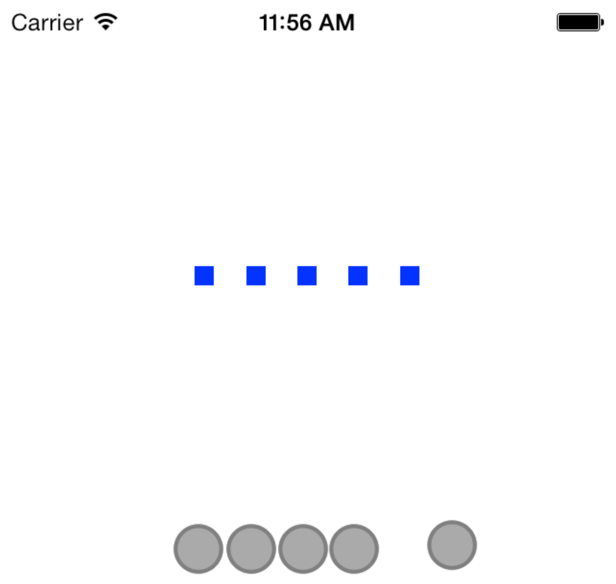
Here we're using a collision behavior and a set of objects which are modeled in the system. Collision behaviors can also be used to model objects hitting boundaries such as view boundaries, or arbitrary bezier path boundaries.

If you run the app now and try to move one of the pendulums you'll notice that the cradle doesn't behave as you would expect it to. This is because the collisions are currently not elastic. We need to add a special type of dynamic behavior to specify various shared properties:

```
UIDynamicItemBehavior *itemBehavior = [[UIDynamicItemBehavior alloc]
                                                  initWithItems:_ballBearings];
// Elasticity governs the efficiency of the collisions
itemBehavior.elasticity = 1.0;
itemBehavior.allowsRotation = NO;
itemBehavior.resistance = 2.0;
[behavior addChildBehavior:itemBehavior];
```

We use `UIDynamicItemBehavior` to specify the elasticity of the collisions, along with some other properties such as resistance (pretty much air resistance) and rotation. If we allow rotation we can specify the angular resistance. The dynamic item behavior also allows setting of linear and angular velocity which can be useful when matching velocities with gestures.

Running the app up now will show a Newton's cradle which behaves exactly as you would expect it in the real world. Maybe as an extension you could investigate drawing the strings of the pendulums as well as the ball bearings.



Completed UIDynamics Newton's Cradle

The code which accompanies this post represents the completed Newton's cradle project. It uses all the elements introduced, but just tidies them up a little into a demo app.

Conclusion

This introduction to UIKit Dynamics has barely scratched the surface - with these building blocks really complex physical systems can be modeled. This opens the door for creating apps which are heavily influenced by our inherent understanding of motion and object interactions from the real world.

Day 1: NSURLSession

In the past networking for iOS was performed using `NSURLConnection` which used the global state to manage cookies and authentication. Therefore it was possible to have 2 different connections competing with each other for shared settings. `NSURLSession` sets out to solve this problem and a host of others as well.

The project which accompanies this guide includes the three different download scenarios discussed forthwith. This post won't describe the entire project - just the salient parts associated with the new `NSURLSession` API.

Simple download

`NSURLSession` represents the entire state associated with multiple connections, which was formerly a shared global state. Session objects are created with a factory method which takes a configuration object. There are 3 types of possible sessions:

1. Default, in-process session
2. Ephemeral (in-memory), in-process session
3. Background session

For a simple download we'll just use a default session:

```
NSURLSessionConfiguration *sessionConfig =
                                [NSURLSessionConfiguration defaultSessionConfiguration];
```

Once a configuration object has been created there are properties on it which control the way it behaves. For example, it's possible to set acceptable levels of TLS security, whether cookies are allowed and timeouts. Two of the more interesting properties are `allowsCellularAccess` and `discretionary`. The former specifies whether a device is permitted to run the networking session when only a cellular radio is available. Setting a session as discretionary enables the operating system to schedule the network access to sensible times - i.e. when a WiFi network is available, and when the device has good power. This is primarily of use for background sessions, and as such defaults to true for a background session.

Once we have a session configuration object we can create the session itself:

```
NSURLSession *inProcessSession;
inProcessSession = [NSURLSession sessionWithConfiguration:sessionConfig
                                                 delegate:self
                                            delegateQueue:nil];
```

Note here that we're also setting ourselves as a delegate. Delegate methods are used to notify us of the progress of data transfers and to request information when challenged for authentication. We'll implement some appropriate methods soon.

Data transfers are encapsulated in tasks - of which there are three types:

1. Data task (`NSURLSessionDataTask`)
2. Upload task (`NSURLSessionUploadTask`)
3. Download task (`NSURLSessionDownloadTask`)

In order to perform a transfer within the session we need to create a task. For a simple file download:

```
NSString *url = @"http://appropriate/url/here";
NSURLRequest *request = [NSURLRequest requestWithURL:[NSURL URLWithString:url]];

NSURLSessionDownloadTask *cancellableTask =
                                      [inProcessSession downloadTaskWithRequest:request];
[cancellableTask resume];
```

That's all there is to it - the session will now asynchronously attempt to download the file at the specified URL.

In order to get hold of the requested file download we need to implement a delegate method:

```
       - (void)URLSession:(NSURLSession *)session
             downloadTask:(NSURLSessionDownloadTask *)downloadTask
didFinishDownloadingToURL:(NSURL *)location
{
    // We've successfully finished the download. Let's save the file
    NSFileManager *fileManager = [NSFileManager defaultManager];

    NSArray *URLs = [fileManager URLsForDirectory:NSDocumentDirectory inDomains:NSUserDoma\
inMask];
    NSURL *documentsDirectory = URLs[0];

    NSURL *destinationPath = [documentsDirectory URLByAppendingPathComponent:
                                                     [location lastPathComponent]];
    NSError *error;

    // Make sure we overwrite anything that's already there
    [fileManager removeItemAtURL:destinationPath error:NULL];
    BOOL success = [fileManager copyItemAtURL:location
                                        toURL:destinationPath
                                        error:&error];

    if (success)
    {
        dispatch_async(dispatch_get_main_queue(), ^{
            UIImage *image = [UIImage imageWithContentsOfFile:[destinationPath path]];
            self.imageView.image = image;
            self.imageView.contentMode = UIViewContentModeScaleAspectFill;
            self.imageView.hidden = NO;
        });
    }
    else
    {
        NSLog(@"Couldn't copy the downloaded file");
    }

    if(downloadTask == cancellableTask) {
        cancellableTask = nil;
    }
}
```

This method is defined on `NSURLSessionDownloadTaskDelegate`. We get passed the temporary location of the downloaded file, so in this code we're saving it off to the documents directory and then (since we have a picture) displaying it to the user.

The above delegate method only gets called if the download task succeeds. The following method is on `NSURLSessionDelegate` and gets called after every task finishes, irrespective of whether it completes successfully:

```
- (void)URLSession:(NSURLSession *)session
              task:(NSURLSessionTask *)task
didCompleteWithError:(NSError *)error
{
    dispatch_async(dispatch_get_main_queue(), ^{
        self.progressIndicator.hidden = YES;
    });
}
```

If the error object is `nil` then the task completed without a problem. Otherwise it's possible to query it to find out what the problem was. If a partial download has been completed then the error object contains a reference to an `NSData` object which can be used to resume the transfer at a later stage.

Tracking progress

You'll have noticed that we hid a progress indicator as part of the task completion method at the end of the last section. Updating the progress of this progress bar couldn't be easier. There is an additional delegate method which is called zero or more times during in the task's lifetime:

```
- (void)URLSession:(NSURLSession *)session
      downloadTask:(NSURLSessionDownloadTask *)downloadTask
      didWriteData:(int64_t)bytesWritten
      BytesWritten:(int64_t)totalBytesWritten
      totalBytesExpectedToWrite:(int64_t)totalBytesExpectedToWrite
{
    double currentProgress = totalBytesWritten / (double)totalBytesExpectedToWrite;
    dispatch_async(dispatch_get_main_queue(), ^{
        self.progressIndicator.hidden = NO;
        self.progressIndicator.progress = currentProgress;
    });
}
```

This is another method which is part of the `NSURLSessionDownloadTaskDelegate`, and we use it here to estimate the progress and update the progress indicator.

Canceling a download

Once an `NSURLConnection` had been sent off it was impossible to cancel it. This is different with an easy ability to cancel the an `NSURLSessionTask`:

```
- (IBAction)cancelCancellable:(id)sender {
    if(cancellableTask) {
        [cancellableTask cancel];
        cancellableTask = nil;
    }
}
```

It's as easy as that! It's worth noting that the `URLSession:task:didCompleteWithError:` delegate method will be called once a task has been canceled to enable you to update the UI appropriately. It's quite possible that after canceling a task the `URLSession:downloadTask:didWriteData:BytesWritten:totalBytesExpectedToWrite:` method might be called again, however, the didComplete method will definitely be last.

Resumable download

It's also possible to resume a download pretty easily. There is an alternative cancel method which provides an `NSData` object which can be used to create a new task to continue the transfer at a later stage. If the server supports resuming downloads then the data object will include the bytes already downloaded:

```
- (IBAction)cancelCancellable:(id)sender {
    if(self.resumableTask) {
        [self.resumableTask cancelByProducingResumeData:^(NSData *resumeData) {
            partialDownload = resumeData;
            self.resumableTask = nil;
        }];
    }
}
```

Here we've popped the resume data into an ivar which we can later use to resume the download.

When creating the download task, rather than supplying a request you can provide a resume data object:

```
if(!self.resumableTask) {
    if(partialDownload) {
        self.resumableTask = [inProcessSession
                                            downloadTaskWithResumeData:partialDownload];
    } else {
        NSString *url = @"http://url/for/image";
        NSURLRequest *request = [NSURLRequest requestWithURL:[NSURL URLWithString:url]];
        self.resumableTask = [inProcessSession downloadTaskWithRequest:request];
    }
    [self.resumableTask resume];
}
```

If we've got a partialDownload object then we create the task using that, otherwise we create the task as we did before.

The only other thing to remember here is that we need to set `partialDownload = nil;` when the process ends.

Background download

The other major feature that `NSURLSession` introduces is the ability to continue data transfers even when your app isn't running. In order to do this we configure a session to be a background session:

```
- (NSURLSession *)backgroundSession
{
    static NSURLSession *backgroundSession = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        NSString *confStr = @"com.shinobicontrols.BackgroundDownload.BackgroundSession"
        NSURLSessionConfiguration *config = [NSURLSessionConfiguration
                                            backgroundSessionConfiguration:confStr];
        backgroundSession = [NSURLSession sessionWithConfiguration:config
```

```
                                                          delegate:self
                                                     delegateQueue:nil];
    });
    return backgroundSession;
}
```

It's important to note that we can only create one session with a given background token, hence the dispatch once block. The purpose of the token is to allow us to collect the session once our app is restarted. Creating a background session starts up a background transfer daemon which will manage the data transfer for us. This will continue to run even when the app has been suspended or terminated.

Starting a background download task is exactly the same as we did before - all of the 'background' functionality is managed by the `NSURLSession` we have just created:

```
NSString *url = @"http://url/for/picture";
NSURLRequest *request = [NSURLRequest requestWithURL:[NSURL URLWithString:url]];
self.backgroundTask = [self.backgroundSession downloadTaskWithRequest:request];
[self.backgroundTask resume];
```

Now, even when you press the home button to leave the app, the download will continue in the background (subject to the configuration options mentioned at the start).

When the download is completed then iOS will restart your app to let it know - and to pass it the payload. To do this it calls the following method on your app delegate:

```
- (void)application:(UIApplication *)application
handleEventsForBackgroundURLSession:(NSString *)identifier
                  completionHandler:(void (^)())completionHandler
{
    self.backgroundURLSessionCompletionHandler = completionHandler;
}
```

Here we get passed a completion handler, which once we've accepted the downloaded data and updated our UI appropriately, we should call. Here we're saving off the completion handler (remembering that blocks have to be copied), and letting the loading of the view controller manage the data handling. When the view controller is loaded it creates the background session (which sets the delegate) and therefore the same delegate methods we were using before are called.

```
- (void)URLSession:(NSURLSession *)session
      downloadTask:(NSURLSessionDownloadTask *)downloadTask
      didFinishDownloadingToURL:(NSURL *)location
{
    // Save the file off as before, and set it as an image view
    //...

    if (session == self.backgroundSession) {
        self.backgroundTask = nil;
        // Get hold of the app delegate
        SCAppDelegate *appDelegate =
                            (SCAppDelegate *)[[UIApplication sharedApplication] delegate];
        if(appDelegate.backgroundURLSessionCompletionHandler) {
            // Need to copy the completion handler
            void (^handler)() = appDelegate.backgroundURLSessionCompletionHandler;
            appDelegate.backgroundURLSessionCompletionHandler = nil;
            handler();
        }
    }
}
```

There are a few things to note here:

- We can’t compare `downloadTask` to `self.backgroundTask`. This is because we can’t guarantee that `self.backgroundTask` has been populated since this could be a new launch of the app. Comparing the session is valid though.
- Here we grab hold of the app delegate. There are other ways of passing the completion handler to the right place.
- Once we’ve finished saving the file and displaying it we make sure that if we have a completion handler, we remove it, and then invoke it. This tells the operating system that we’ve finished handling the new download.

Summary

`NSURLSession` provides a lot of new invaluable features for dealing with networking in iOS (and OSX 10.9) and replaces the old way of doing things. It’s worth getting to grips with it and using it for all apps that can be targetted at the new operating systems.