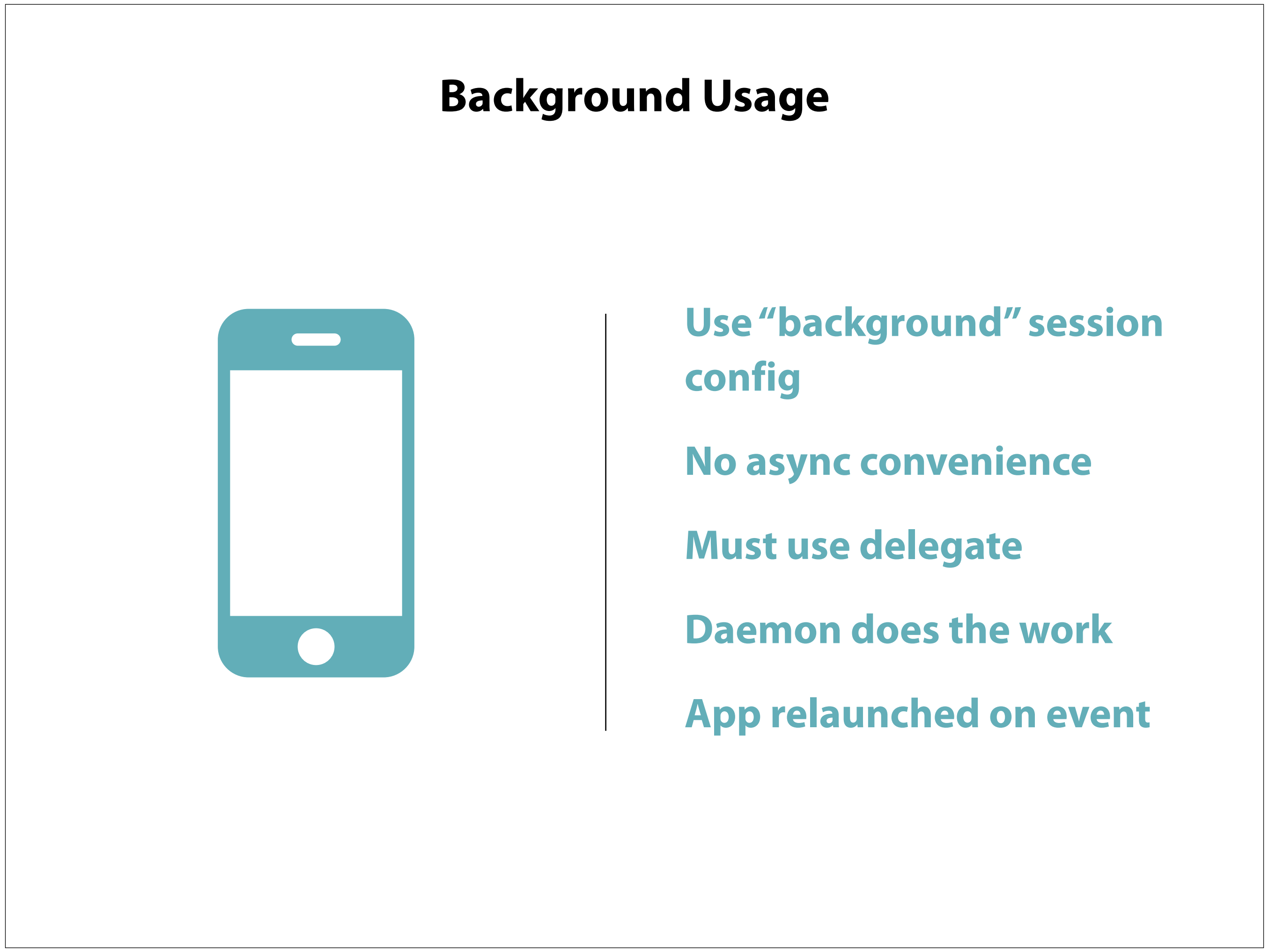
Background Tasks with NSURLSession

Alex Vollmer
http://alexvollmer.com
@alexvollmer

pluralsight
hardcore dev and IT training

One of the key benefits in using NSURLSession is the ability to perform tasks in the background. On iOS this means that the system will handle the connectivity for you, even when your app isn’t running.



Background Usage

Use “background” session config

No async convenience

Must use delegate

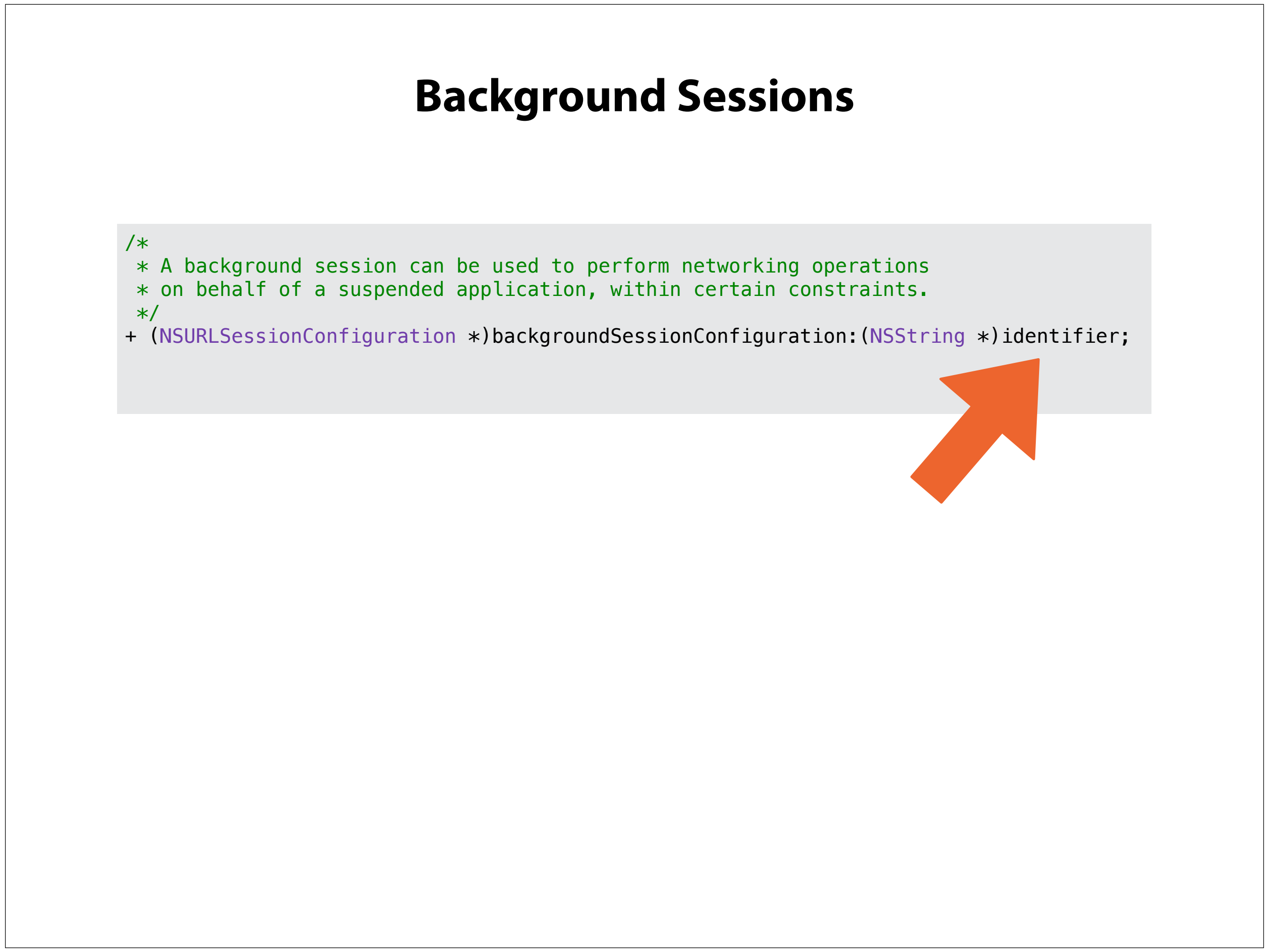
Daemon does the work

App relaunched on event

To take advantage of background processing, you need configure your NSURLSession with a special “background” type of NSURLSessionConfiguration.

You can’t use the asynchronous convenience methods that use block callbacks. Instead you must provide a delegate to the NSURLSession.

The reason for this is that these tasks are handled by an external daemon process. When the daemon has significant events to report, the OS will relaunch your app in the background and deliver the appropriate events.



Background Sessions

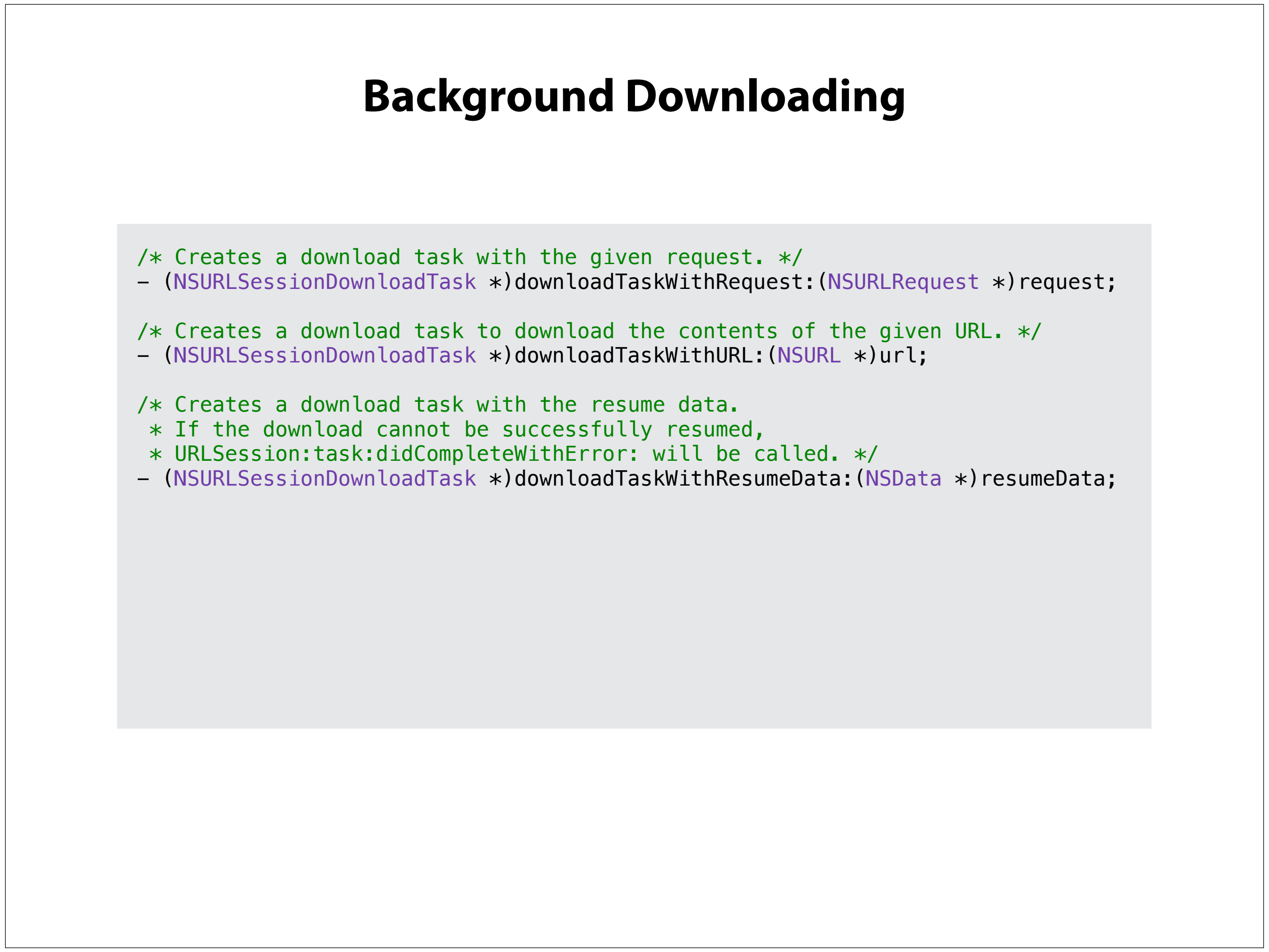
```
/*
 * A background session can be used to perform networking operations
 * on behalf of a suspended application, within certain constraints.
 */
+ (NSURLSessionConfiguration *)backgroundSessionConfiguration:(NSString *)identifier;
```

To launch NSURLSessionTasks in the background, you need to create them from a separate NSURLSession with a "background" session configuration.

You can create a background session configuration with the +backgroundSessionConfiguration factory method which takes a unique string identifier.

If you provide the same string identifier between launches of your application, you can create a NSURLSession that will connect to any outstanding tasks launched with the same identifier.

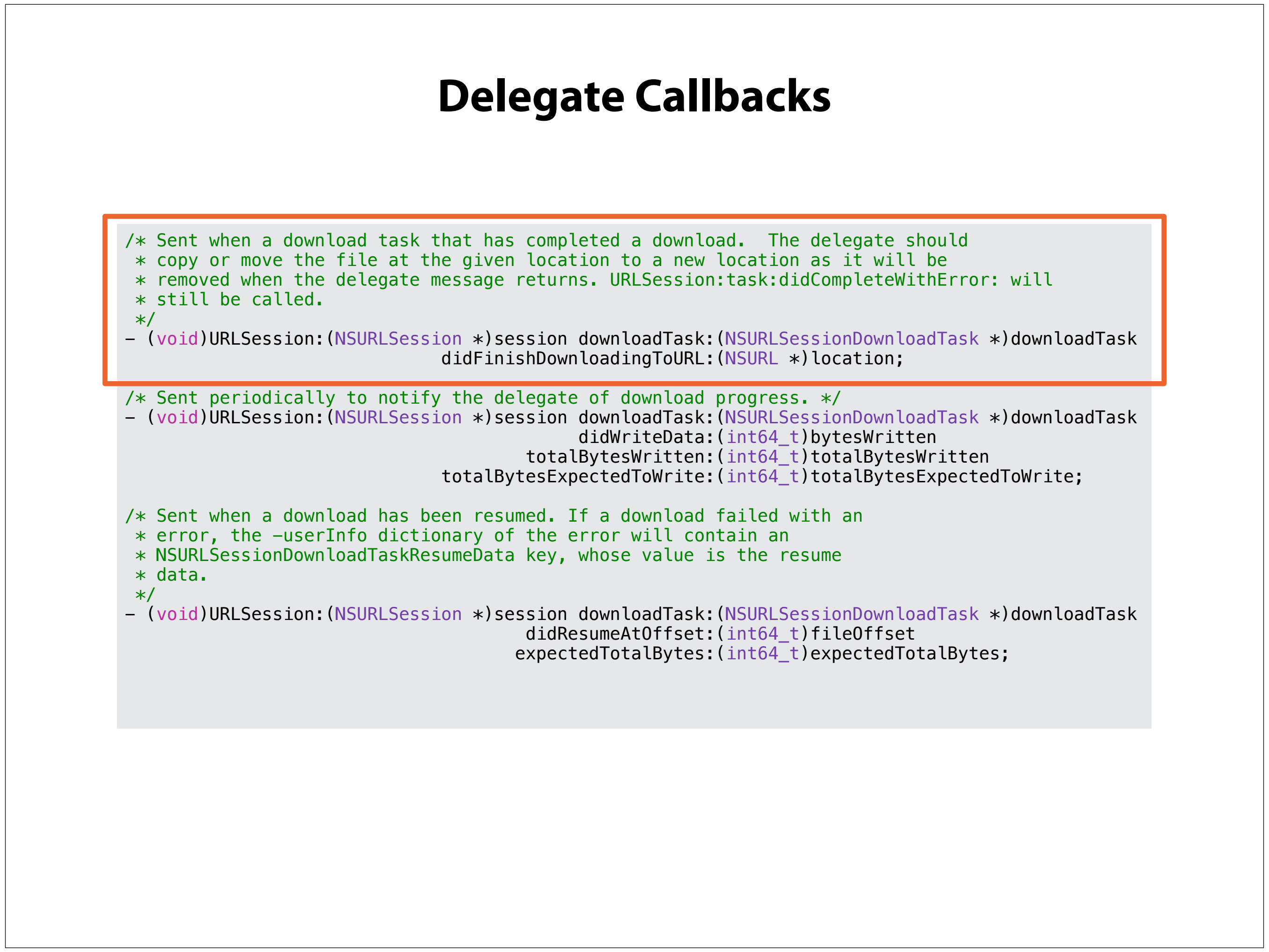
But note that you shouldn't *ever* create two background-configured sessions in the same app with the same identifier.



Background Downloading

```
/* Creates a download task with the given request. */
- (NSURLSessionDownloadTask *)downloadTaskWithRequest:(NSURLRequest *)request;

/* Creates a download task to download the contents of the given URL. */
- (NSURLSessionDownloadTask *)downloadTaskWithURL:(NSURL *)url;

/* Creates a download task with the resume data.
 * If the download cannot be successfully resumed,
 * URLSession:task:didCompleteWithError: will be called. */
- (NSURLSessionDownloadTask *)downloadTaskWithResumeData:(NSData *)resumeData;
```

Once you create a background-enabled NSURLSession, you can create a download task with any of these methods. Remember that you *must* designate a delegate for the session and you *can't* use the asynchronous convenience methods for background operations.



Delegate Callbacks

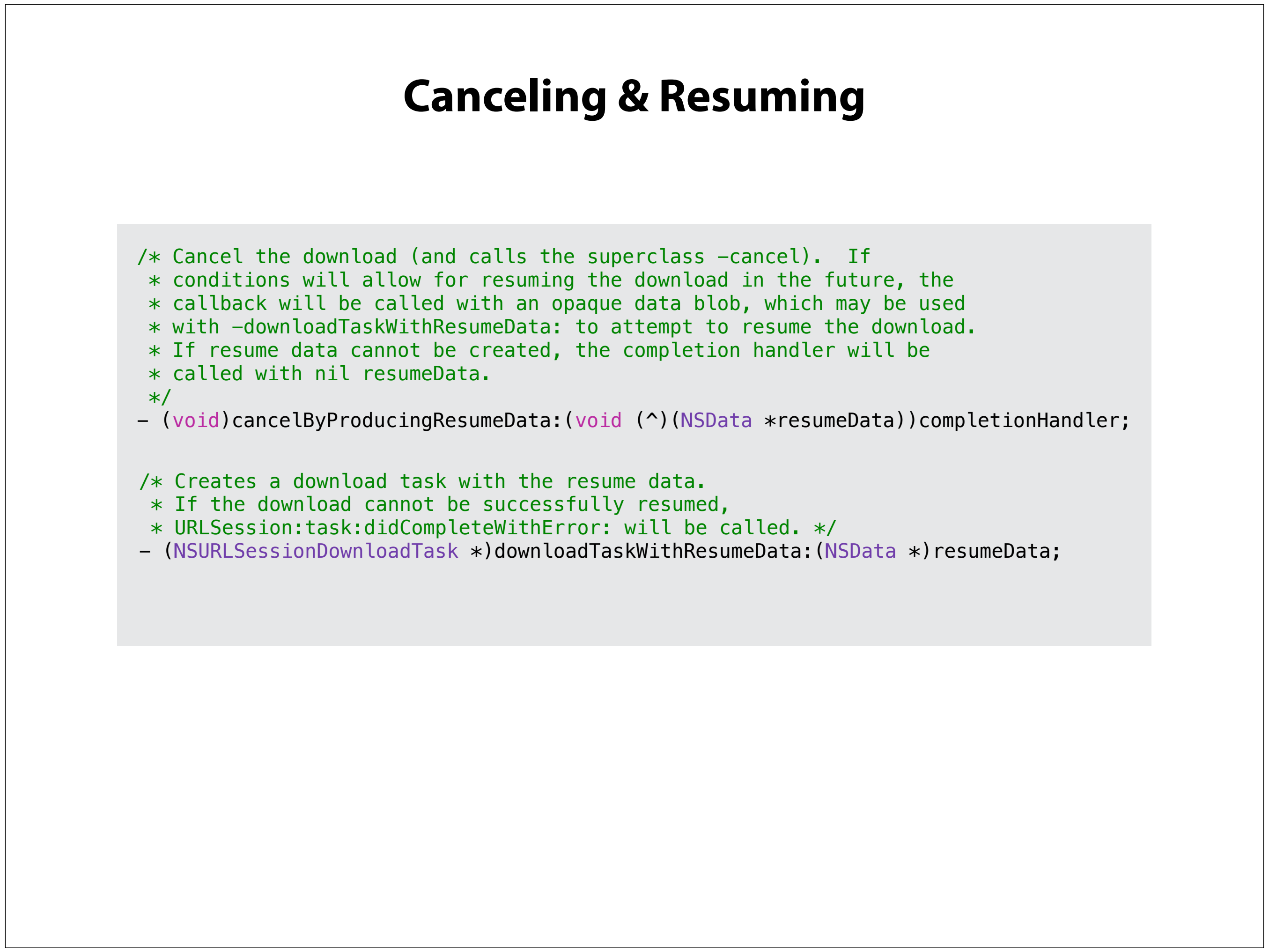
```
/* Sent when a download task that has completed a download.  The delegate should
 * copy or move the file at the given location to a new location as it will be
 * removed when the delegate message returns. URLSession:task:didCompleteWithError: will
 * still be called.
 */
- (void)URLSession:(NSURLSession *)session downloadTask:(NSURLSessionDownloadTask *)downloadTask
                              didFinishDownloadingToURL:(NSURL *)location;

/* Sent periodically to notify the delegate of download progress. */
- (void)URLSession:(NSURLSession *)session downloadTask:(NSURLSessionDownloadTask *)downloadTask
                                           didWriteData:(int64_t)bytesWritten
                                      totalBytesWritten:(int64_t)totalBytesWritten
                              totalBytesExpectedToWrite:(int64_t)totalBytesExpectedToWrite;

/* Sent when a download has been resumed. If a download failed with an
 * error, the -userInfo dictionary of the error will contain an
 * NSURLSessionDownloadTaskResumeData key, whose value is the resume
 * data.
 */
- (void)URLSession:(NSURLSession *)session downloadTask:(NSURLSessionDownloadTask *)downloadTask
                                      didResumeAtOffset:(int64_t)fileOffset
                                     expectedTotalBytes:(int64_t)expectedTotalBytes;
```

As downloading progresses, the NSURLSession's delegate will have these methods called. Note that these methods are *not* optional and *must* be implemented.

The first callback indicates that the download completes successfully. The downloaded file will be in a temporary location that your delegate must move somewhere else because the file will be deleted after this callback is invoked.

As downloading progresses, the -downloadTask:didWriteData:totalyBytesWritten:totalBytesExpectedToWrite: method is called to provide updates to download progress.

If downloading is resumed the third method, -downloadTask:didResumeAtOffset:expectedTotalBytes: will be invoked.



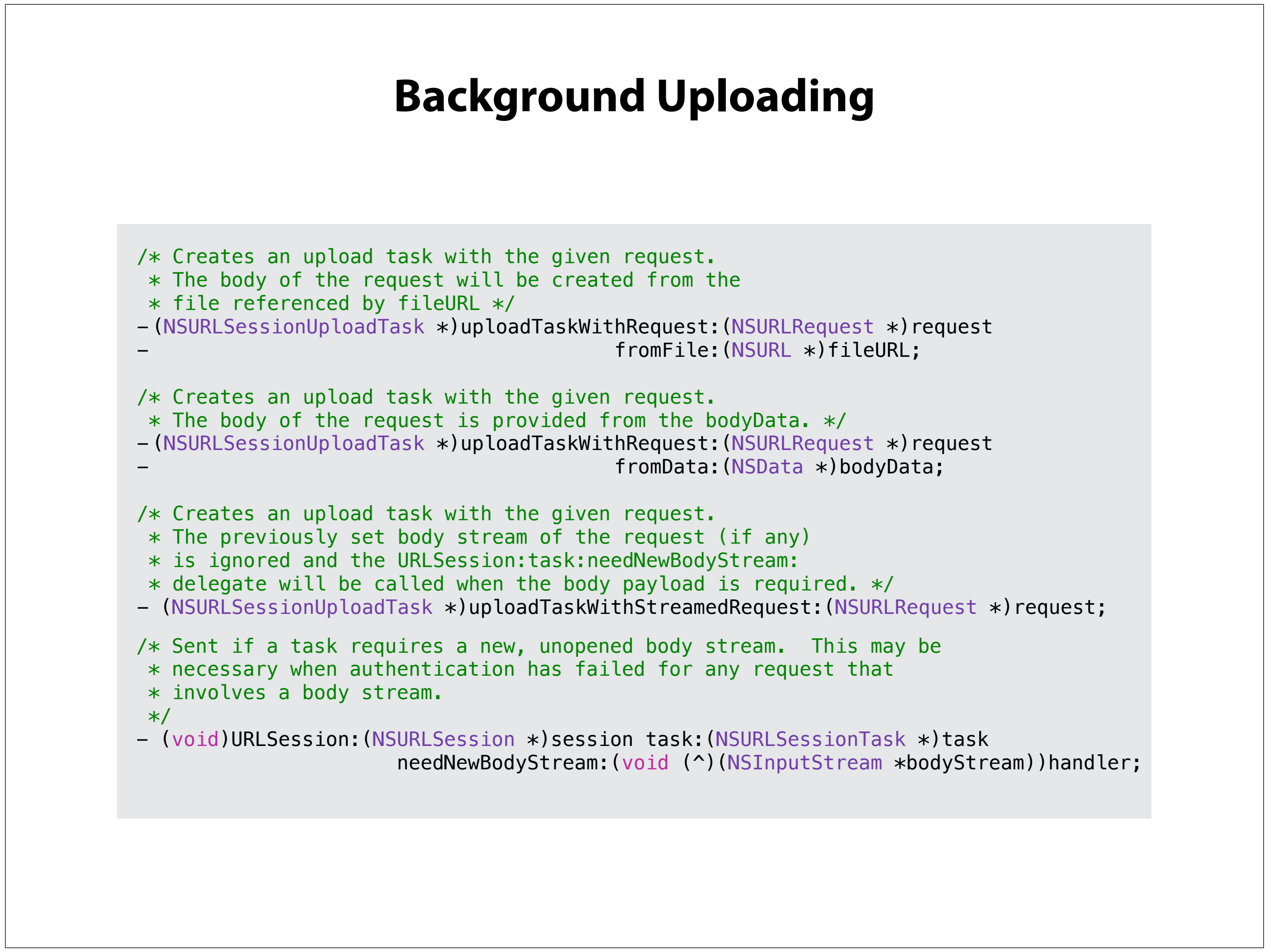
Canceling & Resuming

```
/* Cancel the download (and calls the superclass -cancel).  If
 * conditions will allow for resuming the download in the future, the
 * callback will be called with an opaque data blob, which may be used
 * with -downloadTaskWithResumeData: to attempt to resume the download.
 * If resume data cannot be created, the completion handler will be
 * called with nil resumeData.
 */
- (void)cancelByProducingResumeData:(void (^)(NSData *resumeData))completionHandler;

/* Creates a download task with the resume data.
 * If the download cannot be successfully resumed,
 * URLSession:task:didCompleteWithError: will be called. */
- (NSURLSessionDownloadTask *)downloadTaskWithResumeData:(NSData *)resumeData;
```

If you need to cancel an in-flight download operation, call the -cancelByProducingResumeData: method. This method takes a block which will return an opaque NSData instance to you if the server is able to support resumable downloads. Otherwise the callback block will be give a nil reference.

You can use this NSData object to resume the download operation by calling -downloadTaskWithResumeData: to create a new task to pickup where the old one left off.

One important note is that there's no built-in support for resuming upload tasks. However, if your server supports it, you can manipulate the header fields and uploaded bodies as needed to implement this.

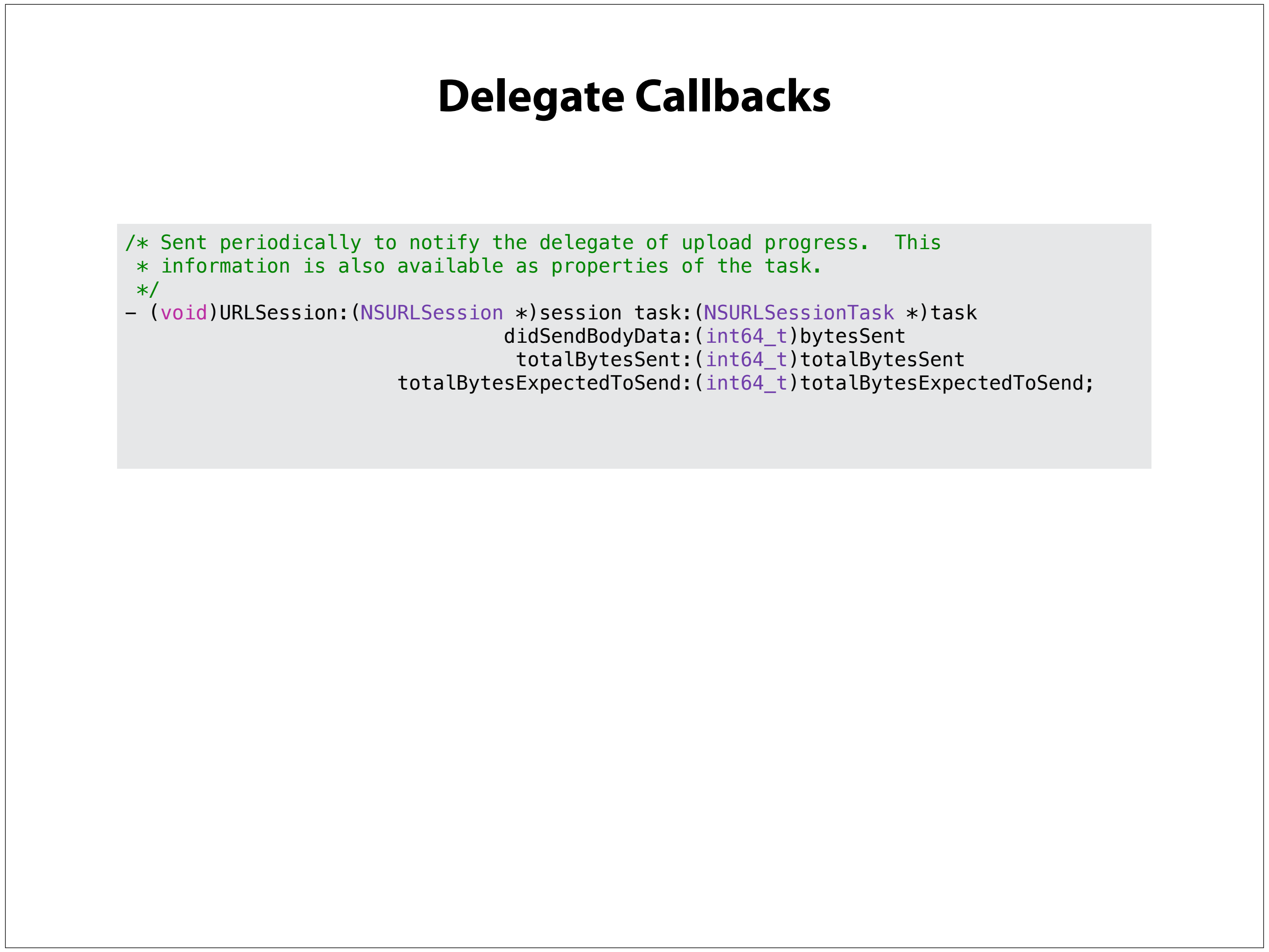


Background Uploading

```
/* Creates an upload task with the given request.
 * The body of the request will be created from the
 * file referenced by fileURL */
-(NSURLSessionUploadTask *)uploadTaskWithRequest:(NSURLRequest *)request
-                                       fromFile:(NSURL *)fileURL;

/* Creates an upload task with the given request.
 * The body of the request is provided from the bodyData. */
-(NSURLSessionUploadTask *)uploadTaskWithRequest:(NSURLRequest *)request
-                                       fromData:(NSData *)bodyData;

/* Creates an upload task with the given request.
 * The previously set body stream of the request (if any)
 * is ignored and the URLSession:task:needNewBodyStream:
 * delegate will be called when the body payload is required. */
- (NSURLSessionUploadTask *)uploadTaskWithStreamedRequest:(NSURLRequest *)request;

/* Sent if a task requires a new, unopened body stream.  This may be
 * necessary when authentication has failed for any request that
 * involves a body stream.
 */
- (void)URLSession:(NSURLSession *)session task:(NSURLSessionTask *)task
                      needNewBodyStream:(void (^)(NSInputStream *bodyStream))handler;
```

To create an upload task for processing in the background, you can call any of these three factory methods. The first two allow you to specify the content to upload directly, while the third defers getting content.

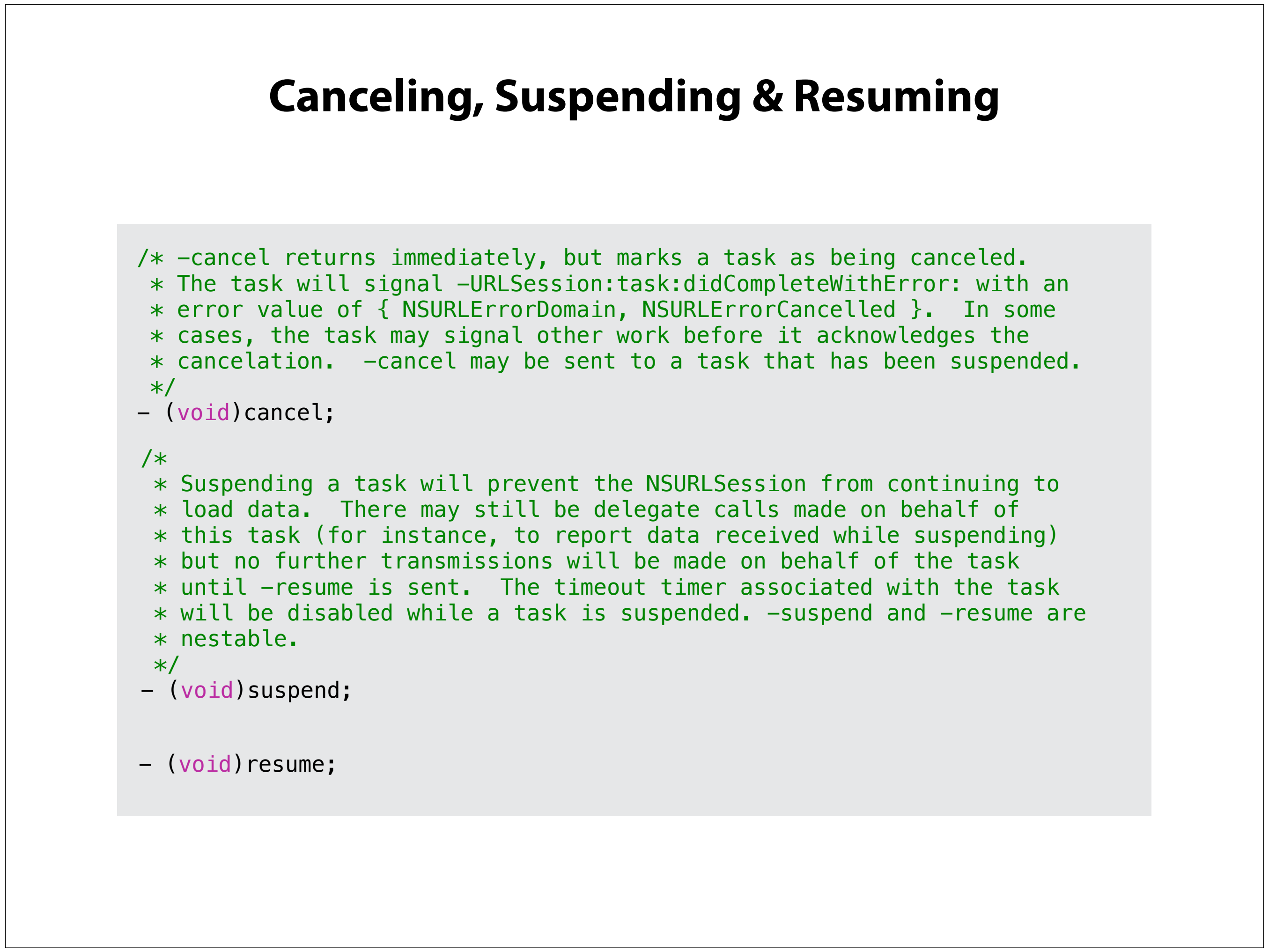
If you create a task with the third method, you must implement the -URLSession:task:needNewBodyStream: delegate method.



Delegate Callbacks

```
/* Sent periodically to notify the delegate of upload progress.  This
 * information is also available as properties of the task.
 */
- (void)URLSession:(NSURLSession *)session task:(NSURLSessionTask *)task
                                didSendBodyData:(int64_t)bytesSent
                                 totalBytesSent:(int64_t)totalBytesSent
                       totalBytesExpectedToSend:(int64_t)totalBytesExpectedToSend;
```

As uploading progresses, a single delegate method, -URLSession:task:didSendBodyData:totalBytesSent:totalBytesExpectedToSend: will be invoked if its implemented.

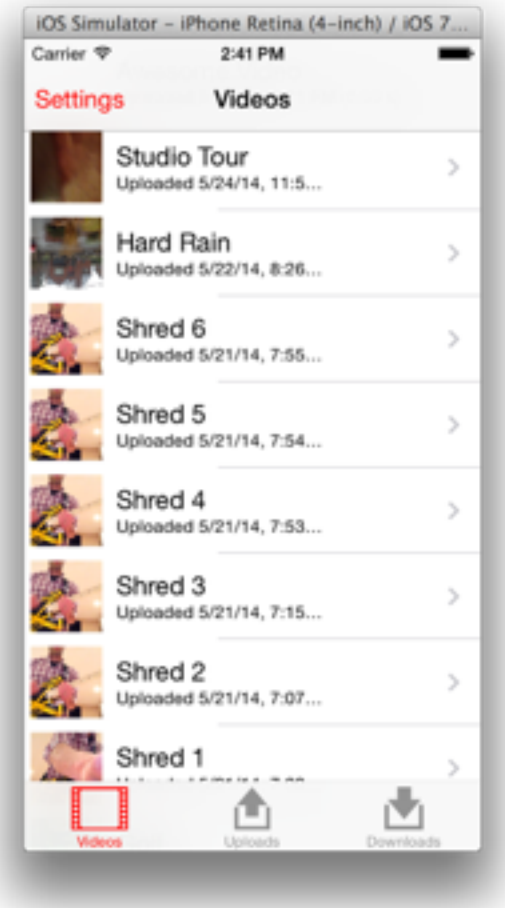


Canceling, Suspending & Resuming

```
/* -cancel returns immediately, but marks a task as being canceled.
 * The task will signal -URLSession:task:didCompleteWithError: with an
 * error value of { NSURLErrorDomain, NSURLErrorCancelled }.  In some
 * cases, the task may signal other work before it acknowledges the
 * cancelation.  -cancel may be sent to a task that has been suspended.
 */
- (void)cancel;

/*
 * Suspending a task will prevent the NSURLSession from continuing to
 * load data.  There may still be delegate calls made on behalf of
 * this task (for instance, to report data received while suspending)
 * but no further transmissions will be made on behalf of the task
 * until -resume is sent.  The timeout timer associated with the task
 * will be disabled while a task is suspended. -suspend and -resume are
 * nestable.
 */
- (void)suspend;


- (void)resume;
```

Like download tasks you can cancel an in-flight upload task. Unlike download operations, you can’t restart a canceled upload operation from where it left off.

However, you can temporarily suspend an NSURLSessionTask with a call to the -suspend method and resume it later by calling the -resume method.



Studio Tour
Uploaded 5/24/14, 11:5...

Hard Rain
Uploaded 5/22/14, 8:26...

Shred 6
Uploaded 5/21/14, 7:55...

Shred 5
Uploaded 5/21/14, 7:54...

Shred 4
Uploaded 5/21/14, 7:53...

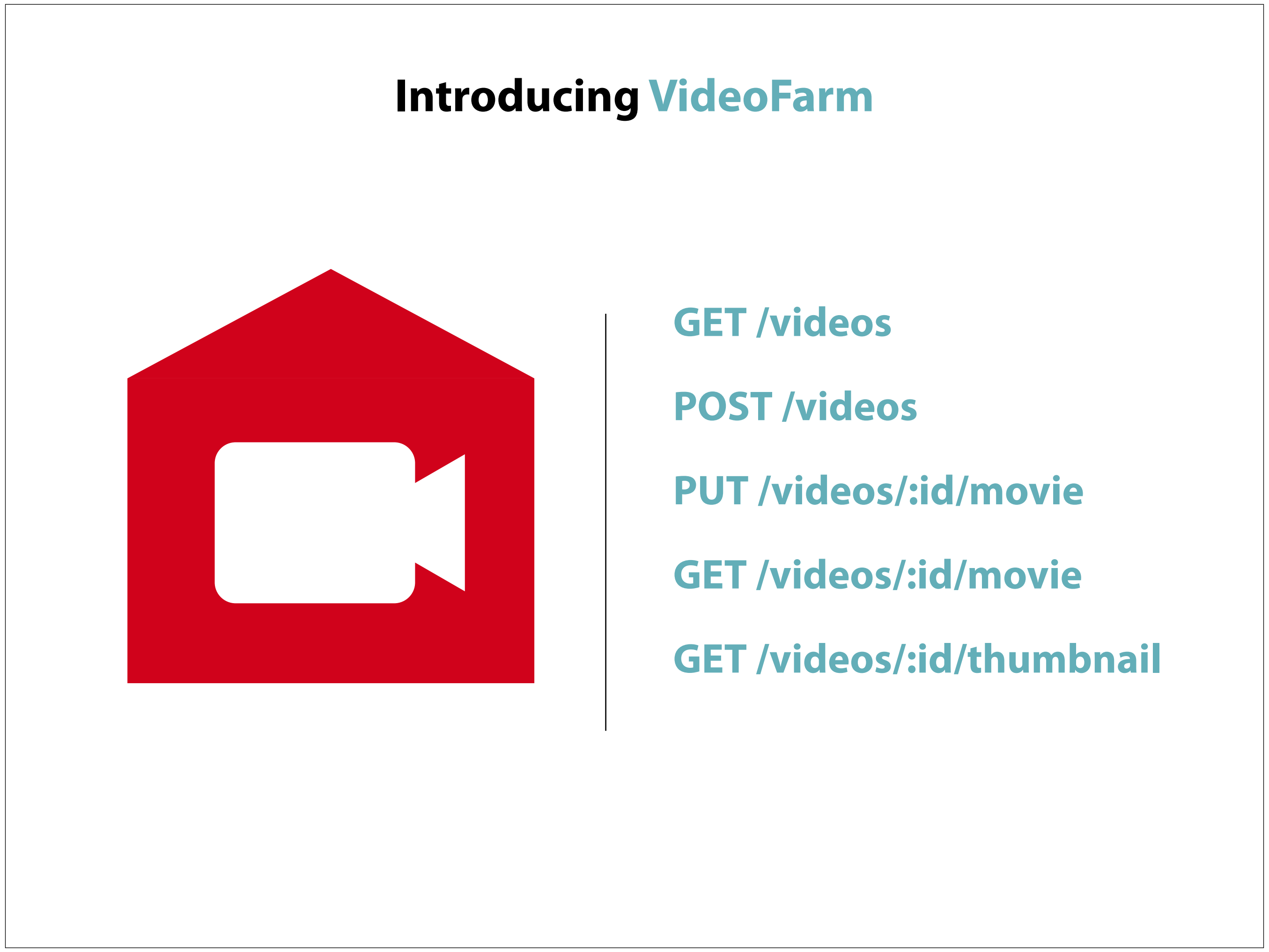
Shred 3
Uploaded 5/21/14, 7:15...

Shred 2
Uploaded 5/21/14, 7:07...

Shred 1

To show you how background uploading and downloading works, we're going to switch to a different sample application, called "VideoFarm".

In this app, can upload, stream and download videos to and from a simple web application. Like ChatCave, I've provided a sample server that's already written in Node.js and you install and run it in the same way.



Introducing VideoFarm

GET /videos

POST /videos

PUT /videos/:id/movie

GET /videos/:id/movie

GET /videos/:id/thumbnail

The VideoFarm server has a very simple REST API.

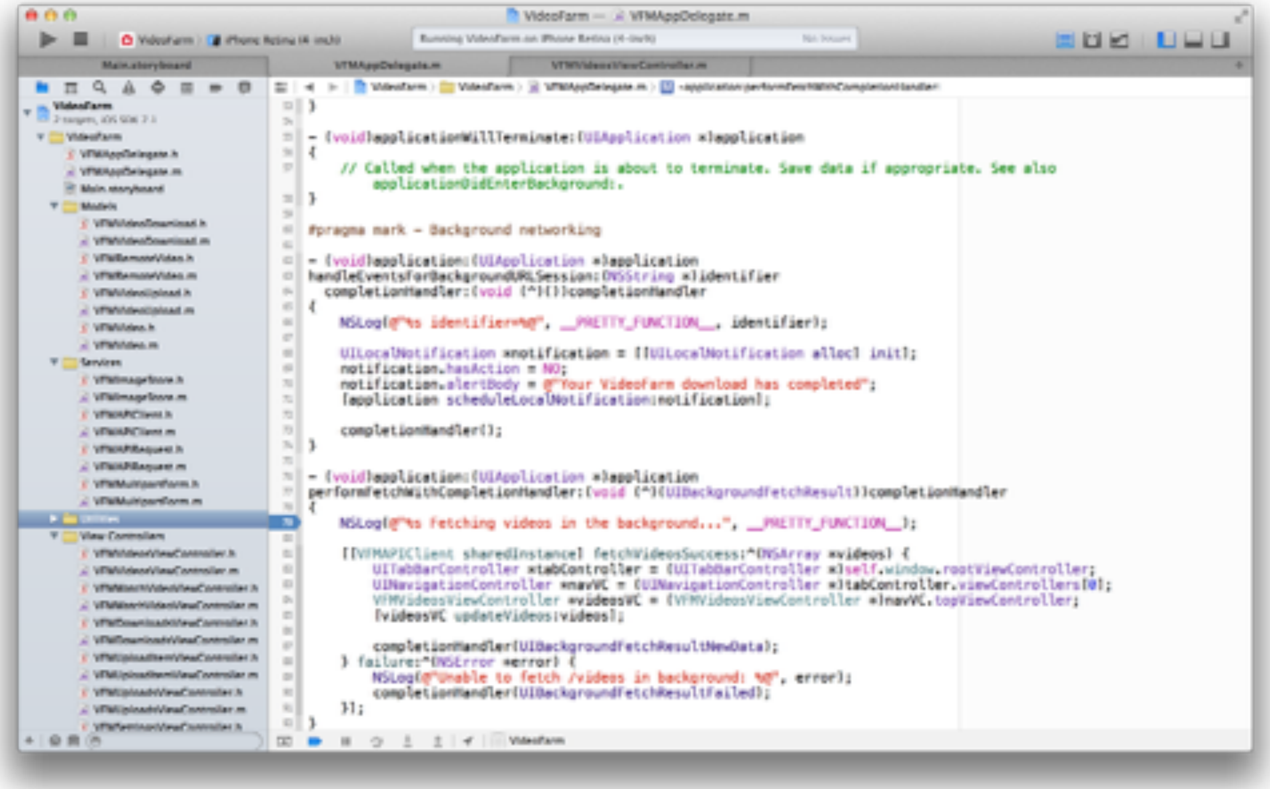
You can retrieve a list of all available videos with a GET request to /videos

The process of adding a new video is split into two resources: you submit the basic metadata (as well as a thumbnail image) by issuing a POST request to /videos. You should receive a 303 response with a Location header indicating where the video lives.

The actual movie file is then submitted by issuing a PUT to /videos/:id/movie

You can also retrieve the movie file from a video with a GET request to /videos/:id/movie.

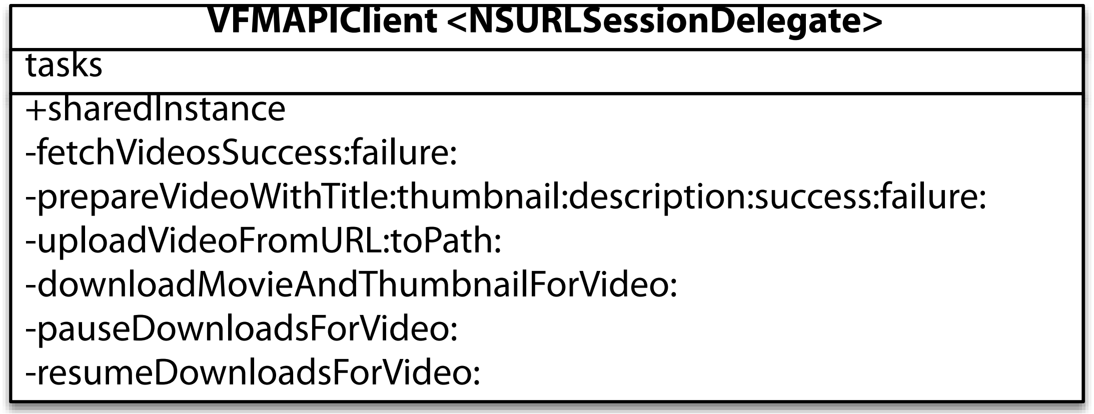
And you can retrieve the thumbnail image with a GET request to /videos/:id/thumbnail


VideoFarm — VFMAppDelegate.m
Running VideoFarm on iPhone Retina (4-inch)
Main.storyboard
VFMAppDelegate.m
VFMVideosViewController.m
VideoFarm
Models
Services
View Controllers

```
}

- (void)applicationWillTerminate:(UIApplication *)application
{
    // Called when the application is about to terminate. Save data if appropriate. See also
        applicationDidEnterBackground:.
}

#pragma mark - Background networking

- (void)application:(UIApplication *)application
handleEventsForBackgroundURLSession:(NSString *)identifier
  completionHandler:(void (^)())completionHandler
{
    NSLog(@"%s identifier=%@", __PRETTY_FUNCTION__, identifier);

    UILocalNotification *notification = [[UILocalNotification alloc] init];
    notification.hasAction = NO;
    notification.alertBody = @"Your Videofarm download has completed";
    [application scheduleLocalNotification:notification];

    completionHandler();
}

- (void)application:(UIApplication *)application
performFetchWithCompletionHandler:(void (^)(UIBackgroundFetchResult))completionHandler
{
    NSLog(@"%s fetching videos in the background...", __PRETTY_FUNCTION__);

    [[VFMAPIClient sharedInstance] fetchVideosSuccess:^(NSArray *videos) {
        UITabBarController *tabController = (UITabBarController *)self.window.rootViewController;
        UINavigationController *navVC = (UINavigationController *)tabController.viewControllers[0];
        VFMVideosViewController *videosVC = (VFMVideosViewController *)navVC.topViewController;
        [videosVC updateVideos:videos];

        completionHandler(UIBackgroundFetchResultNewData);
    } failure:^(NSError *error) {
        NSLog(@"Unable to fetch /videos in background: %@", error);
        completionHandler(UIBackgroundFetchResultFailed);
    }];
}
```

Like the ChatCave server, I've provided sample code that stubs out most of the application. So we don't need to worry about view-controllers or custom views or animations—we'll just focus on the networking code.

VFMAPIClient

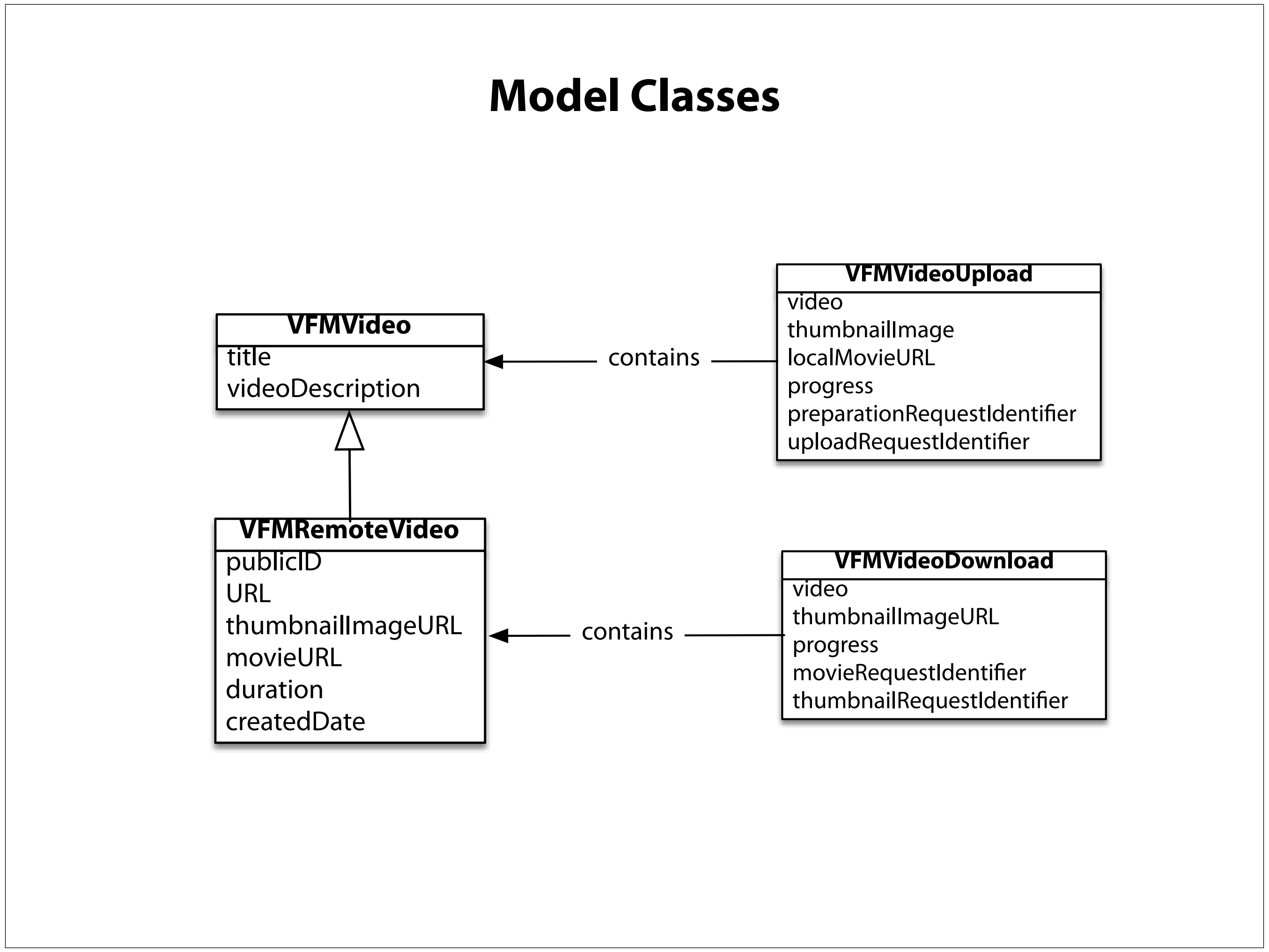


VFMAPIClient <NSURLSessionDelegate>

tasks

+sharedInstance
-fetchVideosSuccess:failure:
-prepareVideoWithTitle:thumbnail:description:success:failure:
-uploadVideoFromURL:toPath:
-downloadMovieAndThumbnailForVideo:
-pauseDownloadsForVideo:
-resumeDownloadsForVideo:

Nearly all of our work will focus on the VFMAPIClient class, which will abstract the network interface to the VideoFarm server.



Model Classes

VFMVideo
title
videoDescription

VFMRemoteVideo
publicID
URL
thumbnailImageURL
movieURL
duration
createdDate

VFMVideoUpload
video
thumbnailImage
localMovieURL
progress
preparationRequestIdentifier
uploadRequestIdentifier

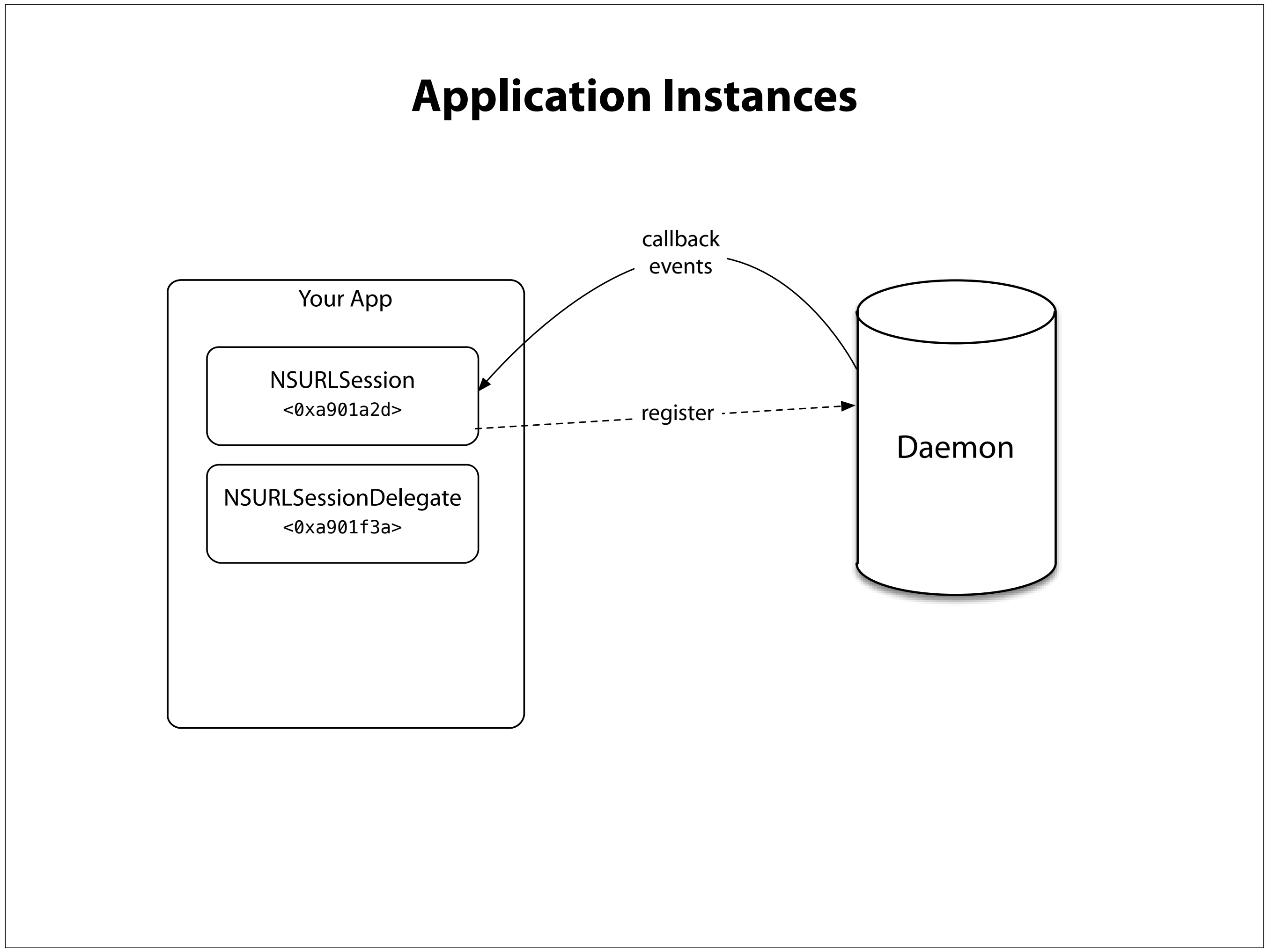
VFMVideoDownload
video
thumbnailImageURL
progress
movieRequestIdentifier
thumbnailRequestIdentifier

There are a handful of model classes we'll be dealing with.

The VFMVideo is a very simple data structure that's extended by VFMRemoteVideo. Instances of this class are created from the JSON we get from the server of existing videos.

The VFMVideoUpload class encapsulates both a VFMVideo as well as additional data associated with the background upload task.

The VFMVideoDownload class encapsulates a VFMRemoteVideo and includes other data needed for managing the background downloading tasks.



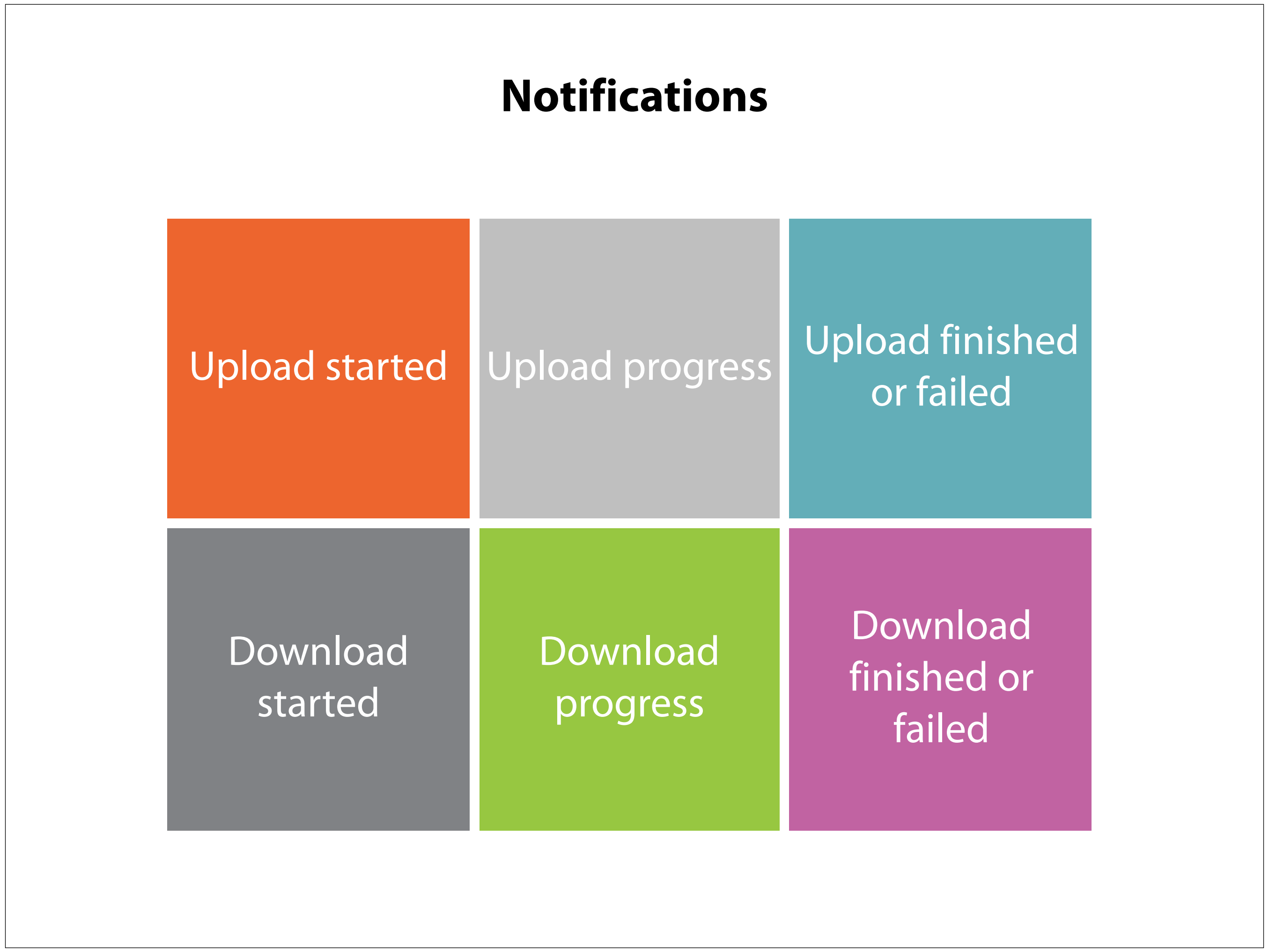
Application Instances

Since background processing happens outside of your app, it can even be killed and still rendezvous with background tasks. But you have to think differently about how your objects will communicate with each other.

Normally, you start the app, create a session, setup its delegate and submit tasks.

When an event occurs, the daemon will callback to the session which will callback into your delegate.

But when your app is killed and restarted, those objects don’t exist anymore. But by using the same identifier for your background session, you can reconnect the graph of objects and get updates.

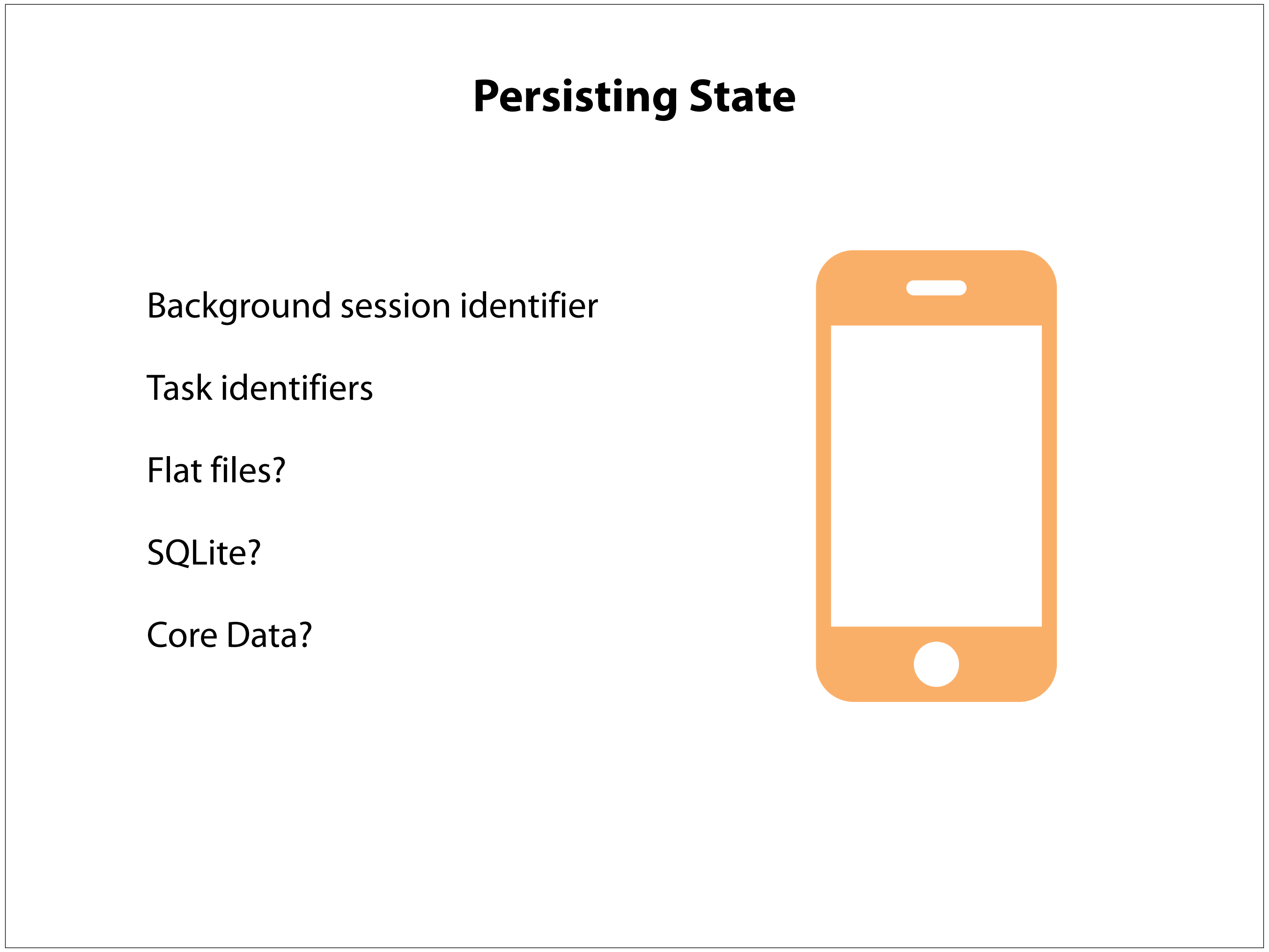


Notifications

Upload started	Upload progress	Upload finished or failed
Download started	Download progress	Download finished or failed

Because of this, the VideoFarm application will rely heavily on notifications to broadcast updates. Using notifications will require less work to reassemble a carefully constructed graph of objects between instances.

Our API client will issue notifications for all of these events which various parts of the application are going to listen for and respond to.



Persisting State

Background session identifier

Task identifiers

Flat files?

SQLite?

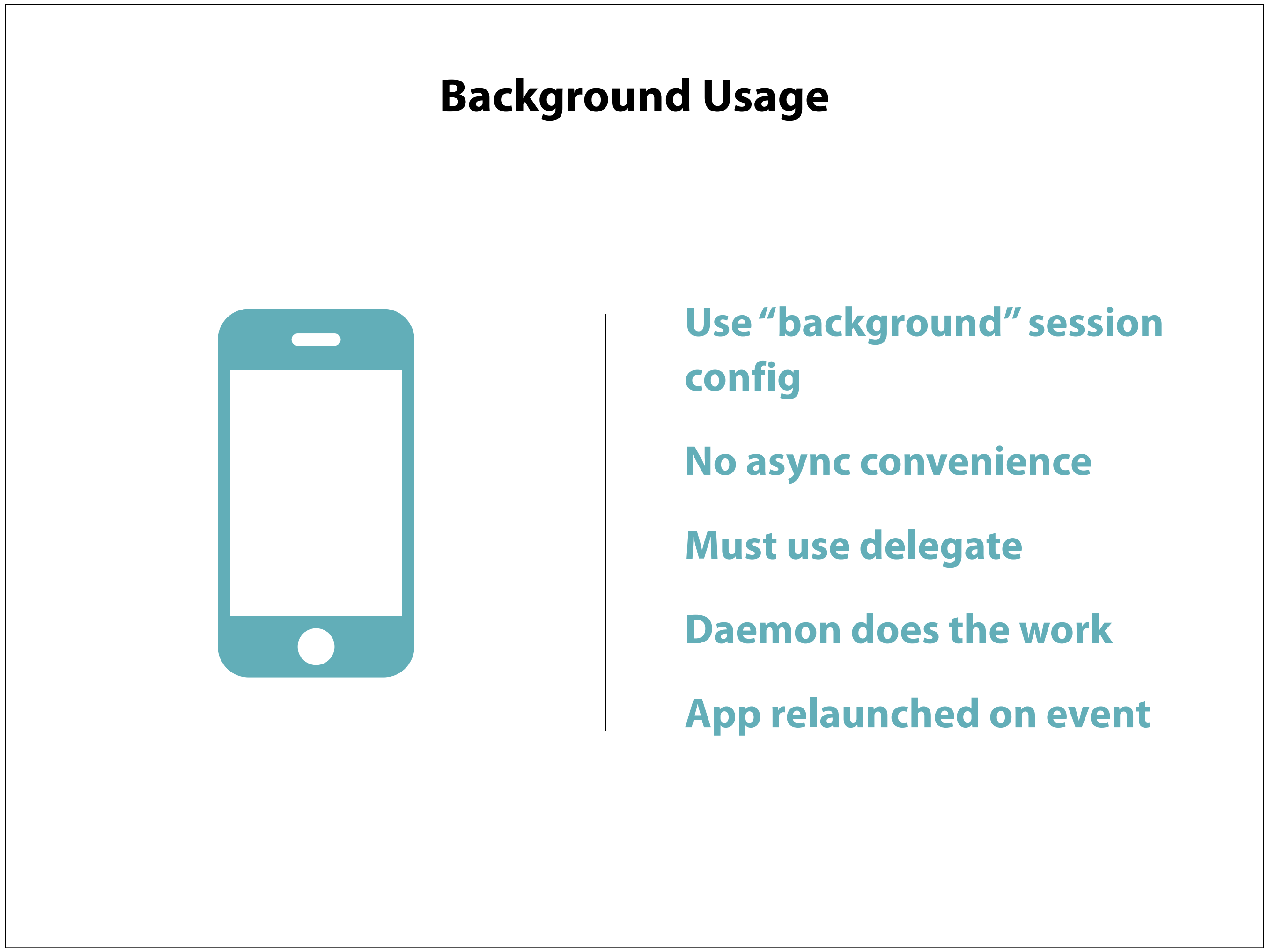
Core Data?

You'll also need to persist some basic state between launches such as the background session identifier, and possibly the task identifiers of any background tasks.

How you do this is entirely up to you. You could use flat-files, a SQL database like SQLite or Core Data.

In the VideoFarm app, we're going to keep it very simple and persist our state to disk in simple archive files.

OK, let's dive in!

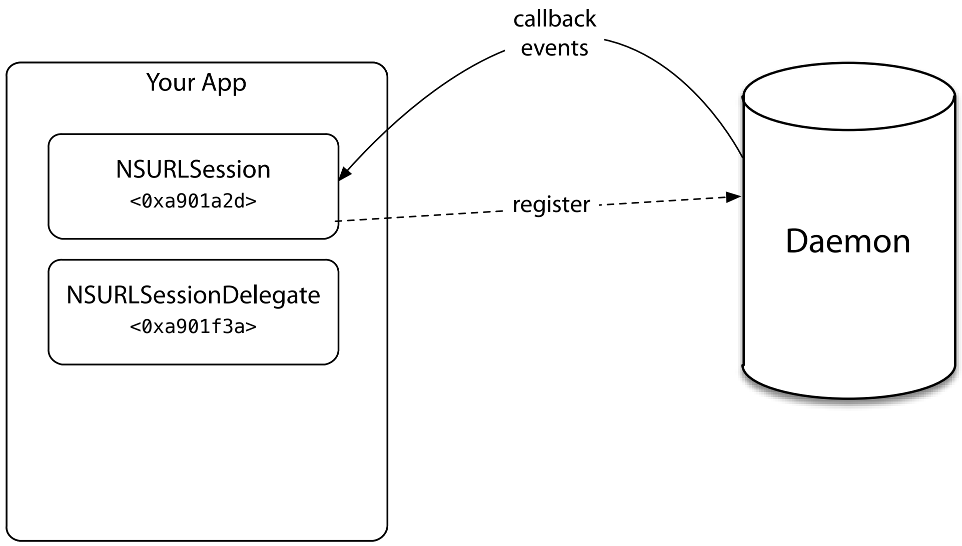


Background Usage

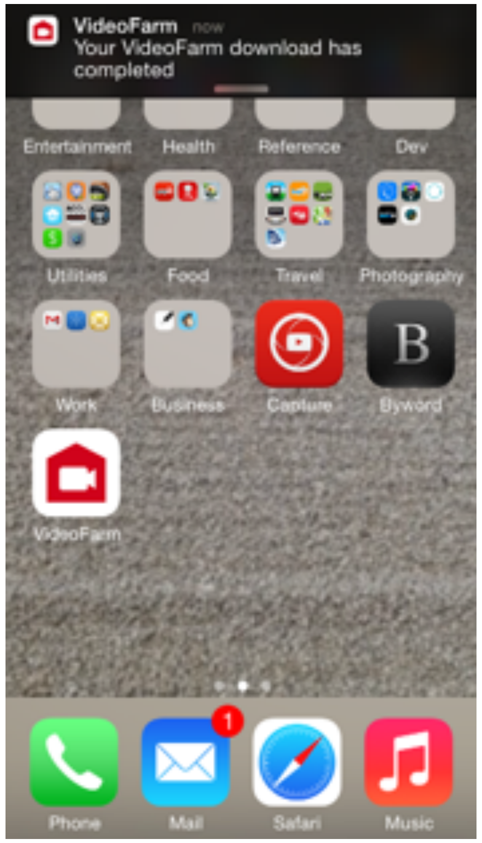
- Use “background” session config
- No async convenience
- Must use delegate
- Daemon does the work
- App relaunched on event

In this module we looked at NSURLSession’s unique ability to process in the background and how uploading and downloading can continue even when your application isn’t running.

Application Instances



We also learned how to reconnect to background tasks between application instances to keep our internal state up-to-date.



VideoFarm now
Your VideoFarm download has completed

We also saw how to improve the user experience by keeping our application’s data as fresh as possible.

We saw how to respond to background tasks completing in the background and how to periodically refresh our data in the background.