Using NSURLSession

The `NSURLSession` class and related classes provide an API for downloading content via HTTP. This API provides a rich set of delegate methods for supporting authentication and gives your app the ability to perform background downloads when your app is not running or, in iOS, while your app is suspended.

To use the `NSURLSession` API, your app creates a series of sessions, each of which coordinates a group of related data transfer tasks. For example, if you are writing a web browser, your app might create one session per tab or window. Within each session, your app adds a series of tasks, each of which represents a request for a specific URL (and for any follow-on URLs if the original URL returned an HTTP redirect).

Like most networking APIs, the `NSURLSession` API is highly asynchronous. If you use the default, system-provided delegate, you must provide a completion handler block that returns data to your app when a transfer finishes successfully or with an error. Alternatively, if you provide your own custom delegate objects, the task objects call those delegates' methods with data as it is received from the server (or, for file downloads, when the transfer is complete).

> **Note:** Completion callbacks are primarily intended as an alternative to using a custom delegate. If you create a task using a method that takes a completion callback, the delegate methods for response and data delivery are not called.

The `NSURLSession` API provides status and progress properties, in addition to delivering this information to delegates. It supports canceling, restarting (resuming), and suspending tasks, and it provides the ability to resume suspended, canceled, or failed downloads where they left off.

Understanding URL Session Concepts

The behavior of the tasks in a session depends on three things: the type of session (determined by the type of configuration object used to create it), the type of task, and whether the app was in the foreground when the task was created.

Types of Sessions

The `NSURLSession` API supports three types of sessions, as determined by the type of configuration object used to create the session:

- *Default sessions* behave similarly to other Foundation methods for downloading URLs. They use a persistent disk-based cache and store credentials in the user's keychain.
- *Ephemeral sessions* do not store any data to disk; all caches, credential stores, and so on are kept in RAM and tied to the session. Thus, when your app invalidates the session, they are purged automatically.
- *Background sessions* are similar to default sessions, except that a separate process handles all data transfers. Background sessions have some additional limitations, described in Background Transfer Considerations.

Types of Tasks

Within a session, the `NSURLSession` class supports three types of tasks: data tasks, download tasks, and upload tasks.

- *Data tasks* send and receive data using `NSData` objects. Data tasks are intended for short, often interactive requests from your app to a server. Data tasks can return data to your app one piece at

a time after each piece of data is received, or all at once through a completion handler.

- *Download tasks* retrieve data in the form of a file, and support background downloads while the app is not running.
- *Upload tasks* send data in the form of a file, and support background uploads while the app is not running.

Background Transfer Considerations

The `NSURLSession` class supports background transfers while your app is suspended. Background transfers are provided only by sessions created using a background session configuration object (as returned by a call to `backgroundSessionConfiguration:`).

With background sessions, because the actual transfer is performed by a separate process and because restarting your app's process is relatively expensive, a few features are unavailable, resulting in the following limitations:

- The session *must* provide a delegate for event delivery. (For uploads and downloads, the delegates behave the same as for in-process transfers.)
- Only HTTP and HTTPS protocols are supported (no custom protocols).
- Redirects are always followed.
- Only upload tasks from a file are supported (uploading from data objects or a stream will fail after the program exits).
- If the background transfer is initiated while the app is in the background, the configuration object's `discretionary` property is treated as being `true`.

> **Note:** Prior to iOS 8 and OS X 10.10, data tasks are not supported in background sessions.

The way your app behaves when it is relaunched differs slightly between iOS and OS X.

In iOS, when a background transfer completes or requires credentials, if your app is no longer running, iOS automatically relaunches your app in the background and calls the `application:handleEventsForBackgroundURLSession:completionHandler:` method on your app's `UIApplicationDelegate` object. This call provides the identifier of the session that caused your app to be launched. Your app should store that completion handler, create a background configuration object with the same identifier, and create a session with that configuration object. The new session is automatically reassociated with ongoing background activity. Later, when the session finishes the last background download task, it sends the session delegate a `URLSessionDidFinishEventsForBackgroundURLSession:` message. In that delegate method, call the previously stored completion handler *on the main thread* so that the operating system knows that it is safe to suspend your app again.

In both iOS and OS X, when the user relaunches your app, your app should immediately create background configuration objects with the same identifiers as any sessions that had outstanding tasks when your app was last running, then create a session for each of those configuration objects. These new sessions are similarly automatically reassociated with ongoing background activity.

> **Note:** You must create *exactly* one session per identifier (specified when you create the configuration object). The behavior of multiple sessions sharing the same identifier is undefined.

If any task completed while your app was suspended, the delegate's `URLSession:downloadTask:didFinishDownloadingToURL:` method is then called with the task and the URL for the newly downloaded file associated with it.

Similarly, if any task requires credentials, the `NSURLSession` object calls the delegate's `URLSession:task:didReceiveChallenge:completionHandler:` method or `URLSession:didReceiveChallenge:completionHandler:` method as appropriate.

Upload and download tasks in background sessions are automatically retried by the URL loading system after network errors. It is unnecessary to use reachability APIs to determine when to retry a failed task.

For an example of how to use `NSURLSession` for background transfers, see *Simple Background Transfer*.

Life Cycle and Delegate Interaction

Depending on what you are doing with the `NSURLSession` class, it may be helpful to fully understand the session life cycle, including how a session interacts with its delegate, the order in which delegate calls are made, what happens when the server returns a redirect, what happens when your app resumes a failed download, and so on.

For a complete description of the life cycle of a URL session, read Life Cycle of a URL Session.

NSCopying Behavior

Session and task objects conform to the `NSCopying` protocol as follows:

- When your app copies a session or task object, you get the same object back.
- When your app copies a configuration object, you get a new copy that you can independently modify.

Sample Delegate Class Interface

The code snippets in the following task sections are based on the class interface shown in Listing 1-1.

Listing 1-1 Sample delegate class interface

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



typedef void (^CompletionHandlerType)();


@interface MySessionDelegate : NSObject <NSURLSessionDelegate,
NSURLSessionTaskDelegate, NSURLSessionDataDelegate, NSURLSessionDownloadDelegate>


@property NSURLSession *backgroundSession;

@property NSURLSession *defaultSession;

@property NSURLSession *ephemeralSession;


#if TARGET_OS_IPHONE

@property NSMutableDictionary *completionHandlerDictionary;

#endif


- (void) addCompletionHandler: (CompletionHandlerType) handler forSession: (NSString
*)identifier;

- (void) callCompletionHandlerForSession: (NSString *)identifier;
```

```
@end
```

Creating and Configuring a Session

The `NSURLSession` API provides a wide range of configuration options:

- Private storage support for caches, cookies, credentials, and protocols in a way that is specific to a single session
- Authentication, tied to a specific request (task) or group of requests (session)
- File uploads and downloads by URL, which encourages separation of the data (the file's contents) from the metadata (the URL and settings)
- Configuration of the maximum number of connections per host
- Per-resource timeouts that are triggered if an entire resource cannot be downloaded in a certain amount of time
- Minimum and maximum TLS version support
- Custom proxy dictionaries
- Control over cookie policies
- Control over HTTP pipelining behavior

Because most settings are contained in a separate configuration object, you can reuse commonly used settings. When you instantiate a session object, you specify the following:

- A configuration object that governs the behavior of that session and the tasks within it
- Optionally, a delegate object to process incoming data as it is received and handle other events specific to the session and the tasks within it, such as server authentication, determining whether a resource load request should be converted into a download, and so on

  If you do not provide a delegate, the `NSURLSession` object uses a system-provided delegate. In this way, you can readily use `NSURLSession` in place of existing code that uses the `sendAsynchronousRequest:queue:completionHandler:` convenience method on `NSURLSession`.

> **Note:** If your app needs to perform background transfers, it must provide a custom delegate.

After you instantiate the session object, you cannot change the configuration or the delegate without creating a new session.

Listing 1-2 shows examples of how to create normal, ephemeral, and background sessions.

Listing 1-2 Creating and configuring sessions

```
#if TARGET_OS_IPHONE
    self.completionHandlerDictionary = [NSMutableDictionary
dictionaryWithCapacity:0];
#endif


    /* Create some configuration objects. */


    NSURLSessionConfiguration *backgroundConfigObject = [NSURLSessionConfiguration
backgroundSessionConfiguration: @"myBackgroundSessionIdentifier"];
```

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

    NSURLSessionConfiguration *ephemeralConfigObject = [NSURLSessionConfiguration
ephemeralSessionConfiguration];



    /* Configure caching behavior for the default session.

       Note that iOS requires the cache path to be a path relative

       to the ~/Library/Caches directory, but OS X expects an

       absolute path.

     */

#if TARGET_OS_IPHONE

    NSString *cachePath = @"/MyCacheDirectory";


    NSArray *myPathList = NSSearchPathForDirectoriesInDomains(NSCachesDirectory,
NSUserDomainMask, YES);

    NSString *myPath    = [myPathList  objectAtIndex:0];


    NSString *bundleIdentifier = [[NSBundle mainBundle] bundleIdentifier];


    NSString *fullCachePath = [[myPath
stringByAppendingPathComponent:bundleIdentifier]
stringByAppendingPathComponent:cachePath];

    NSLog(@"Cache path: %@\n", fullCachePath);

#else

    NSString *cachePath = [NSTemporaryDirectory()
stringByAppendingPathComponent:@"/nsurlsessiondemo.cache"];


    NSLog(@"Cache path: %@\n", cachePath);

#endif









    NSURLCache *myCache = [[NSURLCache alloc] initWithMemoryCapacity: 16384
diskCapacity: 268435456 diskPath: cachePath];

    defaultConfigObject.URLCache = myCache;

    defaultConfigObject.requestCachePolicy = NSURLRequestUseProtocolCachePolicy;


    /* Create a session for each configurations. */

    self.defaultSession = [NSURLSession sessionWithConfiguration:
defaultConfigObject delegate: self delegateQueue: [NSOperationQueue mainQueue]];

    self.backgroundSession = [NSURLSession sessionWithConfiguration:
backgroundConfigObject delegate: self delegateQueue: [NSOperationQueue mainQueue]];

    self.ephemeralSession = [NSURLSession sessionWithConfiguration:
ephemeralConfigObject delegate: self delegateQueue: [NSOperationQueue mainQueue]];
```

With the exception of background configurations, you can reuse session configuration objects to create additional sessions. (You cannot reuse background session configurations because the behavior of two background session objects sharing the same identifier is undefined.)

You can also safely modify the configuration objects at any time. When you create a session, the session performs a deep copy on the configuration object, so modifications affect only new sessions, not existing sessions. For example, you might create a second session for content that should be retrieved only if you are on a Wi-Fi connection as shown in Listing 1-3.

Listing 1-3 Creating a second session with the same configuration object

```
    ephemeralConfigObject.allowsCellularAccess = NO;


    // ...


    NSURLSession *ephemeralSessionWiFiOnly = [NSURLSession sessionWithConfiguration:
ephemeralConfigObject delegate: self delegateQueue: [NSOperationQueue mainQueue]];
```

Fetching Resources Using System-Provided Delegates

The most straightforward way to use the `NSURLSession` is as a drop-in replacement for the `sendAsynchronousRequest:queue:completionHandler:` method on `NSURLConnection`. Using this approach, you need to provide only two pieces of code in your app:

- Code to create a configuration object and a session based on that object
- A completion handler routine to do something with the data after it has been fully received

Using system-provided delegates, you can fetch a specific URL with just a single line of code per request. Listing 1-4 shows an example of this simplified form.

> **Note:** The system-provided delegate provides only limited customization of networking behavior. If your app has special needs beyond basic URL fetching, such as custom authentication or background downloads, this technique is not appropriate. For a complete list of situations in which you must implement a full delegate, see Life Cycle of a URL Session.

Listing 1-4 Requesting a resource using system-provided delegates

```
    NSURLSession *delegateFreeSession = [NSURLSession sessionWithConfiguration:
defaultConfigObject delegate: nil delegateQueue: [NSOperationQueue mainQueue]];


    [[delegateFreeSession dataTaskWithURL: [NSURL URLWithString:
@"http://www.example.com/"]
                        completionHandler:^(NSData *data, NSURLResponse *response,
                                            NSError *error) {
                            NSLog(@"Got response %@ with error %@.\n", response,
error);
                            NSLog(@"DATA:\n%@\nEND DATA\n",
                                  [[NSString alloc] initWithData: data
                                                  encoding: NSUTF8StringEncoding]);
                        }] resume];
```

Fetching Data Using a Custom Delegate

If you are using a custom delegate to retrieve data, the delegate must implement at least the following methods:

- `URLSession:dataTask:didReceiveData:` provides the data from a request to your task, one piece at a time.
- `URLSession:task:didCompleteWithError:`indicates to your task that the data has been fully received.

If your app needs to use the data after its `URLSession:dataTask:didReceiveData:` method returns, your code is responsible for storing the data in some way.

For example, a web browser might need to render the data as it arrives along with any data it has previously received. To do this, it might use a dictionary that maps the task object to an `NSMutableData` object for storing the results, and then use the `appendData:` method on that object to append the newly received data.

Listing 1-5 shows how you create and start a data task.

Listing 1-5 Data task example

```
NSURL *url = [NSURL URLWithString: @"http://www.example.com/"];


NSURLSessionDataTask *dataTask = [self.defaultSession dataTaskWithURL: url];
[dataTask resume];
```

Downloading Files

At a high level, downloading a file is similar to retrieving data. Your app should implement the following delegate methods:

- `URLSession:downloadTask:didFinishDownloadingToURL:` provides your app with the URL to a temporary file where the downloaded content is stored.

  Important: Before this method returns, it must either open the file for reading or move it to a permanent location. When this method returns, the temporary file is deleted if it still exists at its original location.

- `URLSession:downloadTask:didWriteData:totalBytesWritten:totalBytesExpectedToWrite:` provides your app with status information about the progress of the download.
- `URLSession:downloadTask:didResumeAtOffset:expectedTotalBytes:` tells your app that its attempt to resume a previously failed download was successful.
- `URLSession:task:didCompleteWithError:` tells your app that the download failed.

If you schedule the download in a background session, the download continues when your app is not running. If you schedule the download in a standard or ephemeral session, the download must begin anew when your app is relaunched.

During the transfer from the server, if the user tells your app to pause the download, your app can cancel the task by calling the `cancelByProducingResumeData:` method. Later, your app can pass the returned resume data to either the `downloadTaskWithResumeData:` or `downloadTaskWithResumeData:completionHandler:` method to create a new download task that continues the download.

If the transfer fails, your delegate's `URLSession:task:didCompleteWithError:` method is called with an `NSError` object. If the task is resumable, that object's `userInfo` dictionary contains a value for the `NSURLSessionDownloadTaskResumeData` key; your app can pass the returned resume data to either the `downloadTaskWithResumeData:` or

`downloadTaskWithResumeData:completionHandler:` method to create a new download task that retries the download.

Listing 1-6 provides an example of downloading a moderately large file. Listing 1-7 provides an example of download task delegate methods.

Listing 1-6 Download task example

```
    NSURL *url = [NSURL URLWithString:
@"https://developer.apple.com/library/ios/documentation/Cocoa/Reference/"
                  "Foundation/ObjC_classic/FoundationObjC.pdf"];


    NSURLSessionDownloadTask *downloadTask = [self.backgroundSession
downloadTaskWithURL: url];
    [downloadTask resume];
```

Listing 1-7 Delegate methods for download tasks

```
-(void)URLSession:(NSURLSession *)session downloadTask:(NSURLSessionDownloadTask
*)downloadTask didFinishDownloadingToURL:(NSURL *)location
{
    NSLog(@"Session %@ download task %@ finished downloading to URL %@\n",
        session, downloadTask, location);


#if 0
    /* Workaround */
    [self callCompletionHandlerForSession:session.configuration.identifier];
#endif


#define READ_THE_FILE 0
#if READ_THE_FILE
    /* Open the newly downloaded file for reading. */
    NSError *err = nil;
    NSFileHandle *fh = [NSFileHandle fileHandleForReadingFromURL:location
        error: &err];


    /* Store this file handle somewhere, and read data from it. */
    // ...


#else
    NSError *err = nil;
    NSFileManager *fileManager = [NSFileManager defaultManager];
    NSString *cacheDir = [[NSHomeDirectory()
        stringByAppendingPathComponent:@"Library"]
        stringByAppendingPathComponent:@"Caches"];
    NSURL *cacheDirURL = [NSURL fileURLWithPath:cacheDir];
    if ([fileManager moveItemAtURL:location
        toURL:cacheDirURL
        error: &err]) {


        /* Store some reference to the new URL */
```

```
    } else {
        /* Handle the error. */
    }
#endif


}


-(void)URLSession:(NSURLSession *)session downloadTask:(NSURLSessionDownloadTask
*)downloadTask didWriteData:(int64_t)bytesWritten totalBytesWritten:
(int64_t)totalBytesWritten totalBytesExpectedToWrite:
(int64_t)totalBytesExpectedToWrite
{
    NSLog(@"Session %@ download task %@ wrote an additional %lld bytes (total %lld
bytes) out of an expected %lld bytes.\n",
        session, downloadTask, bytesWritten, totalBytesWritten,
totalBytesExpectedToWrite);
}


-(void)URLSession:(NSURLSession *)session downloadTask:(NSURLSessionDownloadTask
*)downloadTask didResumeAtOffset:(int64_t)fileOffset expectedTotalBytes:
(int64_t)expectedTotalBytes
{
    NSLog(@"Session %@ download task %@ resumed at offset %lld bytes out of an
expected %lld bytes.\n",
        session, downloadTask, fileOffset, expectedTotalBytes);
}
```

Uploading Body Content

Your app can provide the request body content for an HTTP POST request in three ways: as an `NSData` object, as a file, or as a stream. In general, your app should:

- Use an `NSData` object if your app already has the data in memory and has no reason to dispose of it.
- Use a file if the content you are uploading exists as a file on disk, if you are doing background transfer, or if it is to your app's benefit to write it to disk so that it can release the memory associated with that data.
- Use a stream if you are receiving the data over a network or are converting existing `NSURLConnection` code that provides the request body as a stream.

Regardless of which style you choose, if your app provides a custom session delegate, that delegate should implement the `URLSession:task:didSendBodyData:totalBytesSent:totalBytesExpectedToSend:` delegate method to obtain upload progress information.

Additionally, if your app provides the request body using a stream, it must provide a custom session delegate that implements the `URLSession:task:needNewBodyStream:` method, described in more detail in Uploading Body Content Using a Stream.

Uploading Body Content Using an NSData Object

To upload body content with an `NSData` object, your app calls either the `uploadTaskWithRequest:fromData:` or `uploadTaskWithRequest:fromData:completionHandler:` method to create an upload task, and provides request body data through the `fromData` parameter.

The session object computes the `Content-Length` header based on the size of the data object.

Your app must provide any additional header information that the server might require—content type, for example—as part of the URL request object.

Uploading Body Content Using a File

To upload body content from a file, your app calls either the `uploadTaskWithRequest:fromFile:` or `uploadTaskWithRequest:fromFile:completionHandler:` method to create an upload task, and provides a file URL from which the task reads the body content.

The session object computes the `Content-Length` header based on the size of the data object. If your app does not provide a value for the `Content-Type` header, the session also provides one.

Your app can provide any additional header information that the server might require as part of the URL request object.

Uploading Body Content Using a Stream

To upload body content using a stream, your app calls the `uploadTaskWithStreamedRequest:` method to create an upload task. Your app provides a request object with an associated stream from which the task reads the body content.

Your app must provide any additional header information that the server might require—content type and length, for example—as part of the URL request object.

In addition, because the session cannot necessarily rewind the provided stream to re-read data, your app is responsible for providing a new stream in the event that the session must retry a request (for example, if authentication fails). To do this, your app provides a `URLSession:task:needNewBodyStream:` method. When that method is called, your app should perform whatever actions are needed to obtain or create a new body stream, and then call the provided completion handler block with the new stream.

> **Note:** Because your app must provide a `URLSession:task:needNewBodyStream:` delegate method if it provides the body through a stream, this technique is incompatible with using a system-provided delegate.

Uploading a File Using a Download Task

To upload body content for a download task, your app must provide either an `NSData` object or a body stream as part of the `NSURLRequest` object provided when it creates the download request.

If you provide the data using a stream, your app must provide a `URLSession:task:needNewBodyStream:` delegate method to provide a new body stream in the event of an authentication failure. This method is described further in Uploading Body Content Using a Stream.

The download task behaves just like a data task except for the way in which the data is returned to your app.

Handling Authentication and Custom TLS Chain Validation

If the remote server returns a status code that indicates authentication is required and if that authentication requires a connection-level challenge (such as an SSL client certificate), `NSURLSession` calls an authentication challenge delegate method.

- For session-level challenges—`NSURLAuthenticationMethodNTLM`, `NSURLAuthenticationMethodNegotiate`, `NSURLAuthenticationMethodClientCertificate`, or `NSURLAuthenticationMethodServerTrust`—the `NSURLSession` object calls the session delegate's `URLSession:didReceiveChallenge:completionHandler:` method. If your app does not provide a session delegate method, the `NSURLSession` object calls the task delegate's `URLSession:task:didReceiveChallenge:completionHandler:` method to handle the challenge.
- For non-session-level challenges (all others), the `NSURLSession` object calls the session delegate's `URLSession:task:didReceiveChallenge:completionHandler:` method to handle the challenge. If your app provides a session delegate and you need to handle authentication, then you must either handle the authentication at the task level or provide a task-level handler that calls the per-session handler explicitly. The session delegate's `URLSession:didReceiveChallenge:completionHandler:` method is *not* called for non-session-level challenges.

Note: Kerberos authentication is handled transparently.

When authentication fails for a task that has a stream-based upload body, the task cannot necessarily rewind and reuse that stream safely. Instead, the `NSURLSession` object calls the delegate's `URLSession:task:needNewBodyStream:` delegate method to obtain a new `NSInputStream` object that provides the body data for the new request. (The session object does not make this call if the task's upload body is provided from a file or an `NSData` object.)

For more information about writing an authentication delegate method for `NSURLSession`, read Authentication Challenges and TLS Chain Validation.

Handling iOS Background Activity

If you are using `NSURLSession` in iOS, your app is automatically relaunched when a download completes. Your app's `application:handleEventsForBackgroundURLSession:completionHandler:` app delegate method is responsible for recreating the appropriate session, storing a completion handler, and calling that handler when the session calls your session delegate's `URLSessionDidFinishEventsForBackgroundURLSession:` method.

Listing 1-8 and Listing 1-9 show examples of these session and app delegate methods, respectively.

Listing 1-8 Session delegate methods for iOS background downloads

```
#if TARGET_OS_IPHONE
-(void)URLSessionDidFinishEventsForBackgroundURLSession:(NSURLSession *)session
{
    NSLog(@"Background URL session %@ finished events.\n", session);

    if (session.configuration.identifier)
        [self callCompletionHandlerForSession: session.configuration.identifier];
}

- (void) addCompletionHandler: (CompletionHandlerType) handler forSession: (NSString
*)identifier
{
```

```
    if ([ self.completionHandlerDictionary objectForKey: identifier]) {
        NSLog(@"Error: Got multiple handlers for a single session identifier.  This
should not happen.\n");
    }

    [ self.completionHandlerDictionary setObject:handler forKey: identifier];
}

- (void) callCompletionHandlerForSession: (NSString *)identifier
{
    CompletionHandlerType handler = [self.completionHandlerDictionary objectForKey:
identifier];

    if (handler) {
        [self.completionHandlerDictionary removeObjectForKey: identifier];
        NSLog(@"Calling completion handler.\n");

        handler();
    }
}
#endif
```

Listing 1-9 App delegate methods for iOS background downloads

```
- (void)application:(UIApplication *)application
handleEventsForBackgroundURLSession:(NSString *)identifier completionHandler:(void
(^)())completionHandler
{
    NSURLSessionConfiguration *backgroundConfigObject = [NSURLSessionConfiguration
backgroundSessionConfiguration: identifier];

    NSURLSession *backgroundSession = [NSURLSession sessionWithConfiguration:
backgroundConfigObject delegate: self.mySessionDelegate delegateQueue:
[NSOperationQueue mainQueue]];

    NSLog(@"Rejoining session %@\n", identifier);

    [ self.mySessionDelegate addCompletionHandler: completionHandler forSession:
identifier];
}
```