URL Loading System Programming Guide

Developer

Contents

Listings

Objective-CSwift

About the URL Loading System

This guide describes the Foundation framework classes available for interacting with URLs and communicating with servers using standard Internet protocols. Together these classes are referred to as the *URL loading system* .

The URL loading system is a set of classes and protocols that allow your app to access content referenced by a URL. At the heart of this technology is the `NSURL` class, which lets your app manipulate URLs and the resources they refer to.

To support that class, the Foundation framework provides a rich collection of classes that let you load the contents of a URL, upload data to servers, manage cookie storage, control response caching, handle credential storage and authentication in app-specific ways, and write custom protocol extensions.

The URL loading system provides support for accessing resources using the following protocols:

- File Transfer Protocol (`ftp://`)
- Hypertext Transfer Protocol (`http://`)
- Hypertext Transfer Protocol with encryption (`https://`)
- Local file URLs (`file:///`)
- Data URLs (`data://`)

It also transparently supports both proxy servers and SOCKS gateways using the user's system preferences.

Important: In addition to the URL loading system, OS X and iOS provide APIs for opening URLs in other applications, such as Safari. These APIs are not described in this document.

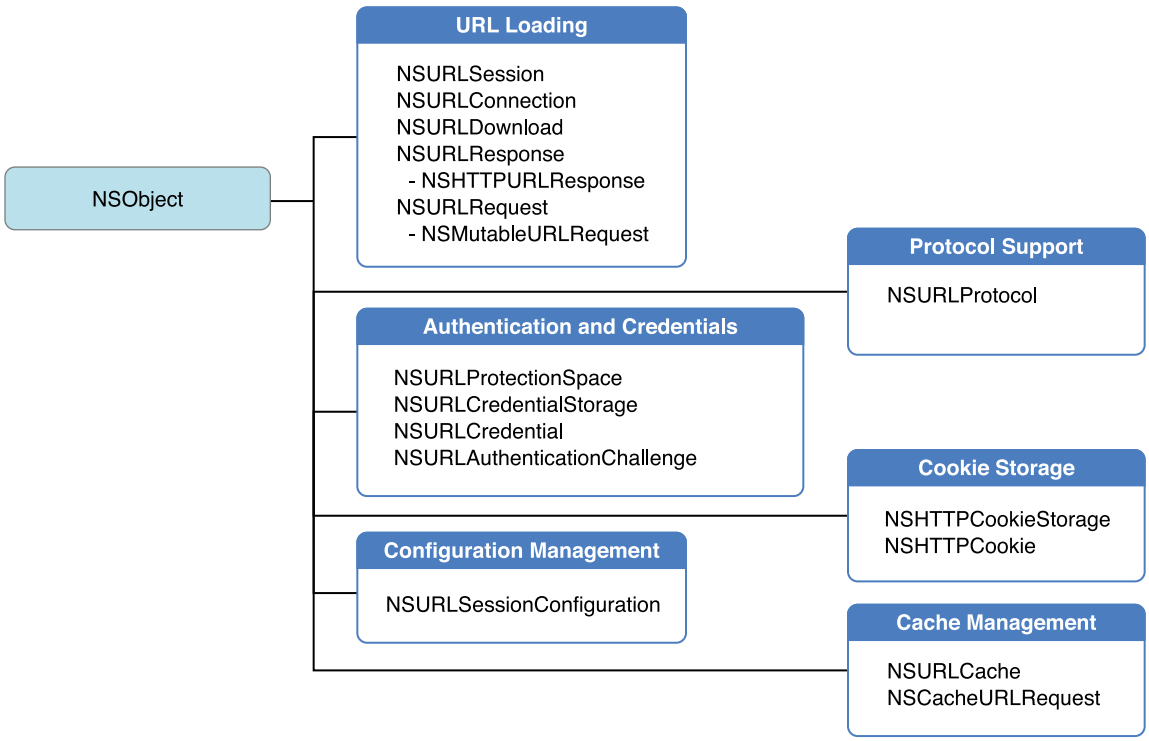
For more information about Launch Services in OS X, read *Launch Services Programming Guide*.

For more information about the `openURL:` method in the `NSWorkSpace` class in OS X, read *NSWorkspace Class Reference*.

For more information about the `openURL:` method in the `UIApplication` class in iOS, read *UIApplication Class Reference*.

At a Glance

The URL loading system includes classes that load URLs along with a number of important helper classes that work with those URL loading classes to modify their behavior. The major helper classes fall into five categories: protocol support, authentication and credentials, cookie storage, configuration management, and cache management.



URL Loading

The most commonly used classes in the URL loading system allow your app to retrieve the content of a URL from the source. You can retrieve that content in many ways, depending on your app's requirements. The API you choose depends on the version of OS X or iOS your app targets and whether you wish to obtain the data as a file or an in-memory block of data:

- In iOS 7 and later or OS X v10.9 and later, `NSURLSession` is the preferred API for new code that performs URL requests.
- For software that must support older versions of OS X, you can use `NSURLDownload` to download the contents of a URL to a file on disk.
- For software that must support older versions of iOS or OS X, you can use `NSURLConnection` to download the contents of a URL to memory. You can then write the data to disk if needed.

The specific methods you use depend largely on whether you wish to fetch data to memory or download it to disk.

Fetching Content as Data (In Memory)

At a high level, there are two basic approaches to fetching URL data:

- For simple requests, use the `NSURLSession` API to retrieve the contents from an `NSURL` object directly, either as an `NSData` object or as a file on disk.
- For more complex requests—requests that upload data, for example—provide an `NSURLRequest` object (or its mutable subclass, `NSMutableURLRequest`) to `NSURLSession` or `NSURLConnection`.

Regardless of which approach you choose, your app can obtain the response data in two ways:

- Provide a completion handler block. The URL loading class calls that block when it finishes receiving data from the server.
- Provide a custom delegate. The URL loading class periodically calls your delegate methods as it receives the data from the originating source. Your app is responsible for accumulating that data, if needed.

In addition to the data itself, the URL loading class provides your delegate or completion handler block with a response object that encapsulates metadata associated with the request, such as the MIME type and content length.

Relevant Chapters: Using NSURLSession (page 16), Using NSURLConnection (page 32)

Downloading Content as a File

At a high level, there are two basic approaches to downloading the contents of a URL to a file:

- For simple requests, use the `NSURLSession` API to retrieve the contents from an `NSURL` object directly, either as an `NSData` object or as a file on disk.
- For more complex requests—requests that upload data, for example—provide an `NSURLRequest` object (or its mutable subclass, `NSMutableURLRequest`) to `NSURLSession` or `NSURLDownload`.

The `NSURLSession` class provides two significant advantages over the `NSURLDownload` class: it is available in iOS, and downloads can continue in the background while your app is suspended, terminated, or crashed.

Note: Downloads initiated by an `NSURLDownload` or `NSURLSession` instance are not cached. If you need to cache the results, your app must use either `NSURLConnection` or `NSURLSession` and write the data to disk itself.

Relevant Chapters: Using NSURLSession (page 16), Using NSURLDownload (page 40)

Helper Classes

The URL loading classes use two helper classes that provide additional metadata—one for the request itself (`NSURLRequest`) and one for the server's response (`NSURLResponse`).

URL Requests

An `NSURLRequest` object encapsulates a URL and any protocol-specific properties, in a protocol-independent manner. It also specifies the policy regarding the use of any locally cached data, and when used with `NSURLConnection` or `NSURLDownload`, provides an interface to set the connection timeout. (For `NSURLSession`, timeouts are configured on a per-session basis.)

> **Note:** When a client app initiates a connection or download using an instance of `NSMutableURLRequest`, a deep copy is made of the request. Changes made to the initiating request have no effect after a download is initialized.

Some protocols support protocol-specific properties. For example, the HTTP protocol adds methods to `NSURLRequest` that return the HTTP request body, headers, and transfer method. It also adds methods to `NSMutableURLRequest` to set those values.

The details of working with URL request objects are described throughout this book.

Response Metadata

The response from a server to a request can be viewed as two parts: metadata describing the contents and the content data itself. Metadata that is common to most protocols is encapsulated by the `NSURLResponse` class and consists of the MIME type, expected content length, text encoding (where applicable), and the URL that provided the response. Protocol-specific subclasses of `NSURLResponse` can provide additional metadata. For example, `NSHTTPURLResponse` stores the headers and the status code returned by the web server.

> **Important:** Only the metadata for the response is stored in an `NSURLResponse` object. The various URL loading classes provide the response data itself to your app either through a completion handler block or to the object's delegate.
>
> An `NSCachedURLResponse` instance encapsulates an `NSURLResponse` object, the URL content data, and any additional information provided by your app. See Cache Management (page 13) for details.

The details of working with URL response objects are described throughout this book.

Redirection and Other Request Changes

Some protocols, such as HTTP, provide a way for a server to tell your app that content has moved to a different URL. The URL loading classes can notify their delegates when this happens. If your app provides an appropriate delegate method, your app can then decide whether to follow the redirect, return the response body from the redirect, or return an error.

Relevant Chapter: Handling Redirects and Other Request Changes (page 50)

Authentication and Credentials

Some servers restrict access to certain content, requiring a user to authenticate by providing some sort of credentials—a client certificate, a user name and password, and so on—in order to gain access. In the case of a web server, restricted content is grouped into a realm that requires a single set of credentials. Credentials (certificates, specifically) are also used to determine trust in the other direction—to evaluate whether your app should trust the server.

The URL loading system provides classes that model credentials and protected areas as well as providing secure credential persistence. Your app can specify that these credentials persist for a single request, for the duration of an app's launch, or permanently in the user's keychain.

Note: Credentials stored in persistent storage are kept in the user's keychain and shared among all apps.

The `NSURLCredential` class encapsulates a credential consisting of authentication information (user name and password, for example) and persistence behavior. The `NSURLProtectionSpace` class represents an area that requires a specific credential. A protection space can be limited to a single URL, encompass a realm on a web server, or refer to a proxy.

A shared instance of the `NSURLCredentialStorage` class manages credential storage and provides the mapping of an `NSURLCredential` object to the corresponding `NSURLProtectionSpace` object for which it provides authentication.

The `NSURLAuthenticationChallenge` class encapsulates the information required by an `NSURLProtocol` implementation to authenticate a request: a proposed credential, the protection space involved, the error or response that the protocol used to determine that authentication is required, and the number of authentication attempts that have been made. An `NSURLAuthenticationChallenge` instance also specifies the object that initiated the authentication. The initiating object, referred to as the *sender*, must conform to the `NSURLAuthenticationChallengeSender` protocol.

`NSURLAuthenticationChallenge` instances are used by `NSURLProtocol` subclasses to inform the URL loading system that authentication is required. They are also provided to the delegate methods of `NSURLConnection` and `NSURLDownload` that facilitate customized authentication handling.

Relevant Chapter: Authentication Challenges and TLS Chain Validation (page 53)

Cache Management

The URL loading system provides a composite on-disk and in-memory cache allowing an app to reduce its dependence on a network connection and provide faster turnaround for previously cached responses. The cache is stored on a per-app basis. The cache is queried by `NSURLConnection` according to the cache policy specified by the initiating `NSURLRequest` object.

The `NSURLCache` class provides methods to configure the cache size and its location on disk. It also provides methods to manage the collection of `NSCachedURLResponse` objects that contain the cached responses.

An `NSCachedURLResponse` object encapsulates the `NSURLResponse` object and the URL content data. `NSCachedURLResponse` also provides a user info dictionary that your app can use to cache any custom data.

Not all protocol implementations support response caching. Currently only `http` and `https` requests are cached.

An `NSURLConnection` object can control whether a response is cached and whether the response should be cached only in memory by implementing the `connection:willCacheResponse:` delegate method.

Relevant Chapter: Understanding Cache Access (page 58)

Cookie Storage

Due to the stateless nature of the HTTP protocol, clients often use cookies to provide persistent storage of data across URL requests. The URL loading system provides interfaces to create and manage cookies, to send cookies as part of an HTTP request, and to receive cookies when interpreting a web server's response.

OS X and iOS provide the `NSHTTPCookieStorage` class, which in turn provides the interface for managing a collection of `NSHTTPCookie` objects. In OS X, cookie storage is shared across all apps; in iOS, cookie storage is per-app.

Relevant Chapter: Cookie Storage (page 62)

Protocol Support

The URL loading system natively supports the `http`, `https`, `file`, `ftp`, and `data` protocols. However, the URL loading system also allows your app to register your own classes to support additional application-layer networking protocols. You can also add protocol-specific properties to URL request and URL response objects.

Relevant Chapter: Cookies and Custom Protocols (page 62)

How to Use This Document

This document is largely divided based on which URL loading class the chapter describes. To decide which API to use, read URL Loading (page 9). After you decide which API you want to use, read the appropriate API-specific chapter or chapters:

- For using the `NSURLSession` class to asynchronously fetch the contents of a URL to memory or download files to disk, read Using NSURLSession (page 16). Then read Life Cycle of a URL Session (page 64) to learn in detail how `NSURLSession` interacts with its delegates.
- For using `NSURLConnection` to asynchronously fetch the contents of a URL to memory, read Using NSURLConnection (page 32).
- For using `NSURLDownload` to download files asynchronously to disk, read Using NSURLDownload (page 40).

After reading one or more of these API-specific chapters, you should also read the following chapters, which are relevant to all three APIs:

- Encoding URL Data (page 48) explains how to encode arbitrary strings to make them safe for use in a URL.
- Handling Redirects and Other Request Changes (page 50) describes the options you have for responding to a change to your URL request.
- Authentication Challenges and TLS Chain Validation (page 53) describes the process for authenticating your connection against a secure server.
- Understanding Cache Access (page 58) describes how a connection uses the cache during a request.
- Cookies and Custom Protocols (page 62) explains the classes available for managing cookie storage and supporting custom application-layer protocols.

See Also

The following sample code is available:

- *LinkedImageFetcher* (OS X) and *AdvancedURLConnections* (iOS) use `NSURLConnection` with custom authentication.
- *SpecialPictureProtocol* (OS X) and *CustomHTTPProtocol* (iOS) show how to implement a custom `NSURLProtocol` subclass.

- *QuickLookDownloader* (in the Mac Developer Library) uses `NSURLDownload` to manage file downloads from the Internet.

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 (page 17).

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. Because data tasks do not store the data to a file, they are not supported in background sessions.
- *Download tasks* retrieve data in the form of a file, and support background downloads while the app is not running.
- *Upload tasks* send data (usually 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).
- Only upload and download tasks are supported (no data tasks).
- Redirects are always followed.
- If the background transfer is initiated while the app is in the background, the configuration object's `discretionary` property is treated as being `true`.

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. Your session delegate should then call the stored completion handler.

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.

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 (page 64).

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 = YES;

    // ...

   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 `NSURLSession`. Using this approach, your 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 (page 23) 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 should use reachability APIs to determine when to retry, and should then call `downloadTaskWithResumeData:` or `downloadTaskWithResumeData:completionHandler:` to create a new download task to continue that 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 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 (page 28).

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 (page 28).

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 (page 53).

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];
}
```

Using NSURLConnection

`NSURLConnection` provides the most flexible method of retrieving the contents of a URL. This class provides a simple interface for creating and canceling a connection, and supports a collection of delegate methods that provide feedback and control of many aspects of the connection. These classes fall into five categories: URL loading, cache management, authentication and credentials, cookie storage, and protocol support.

Creating a Connection

The `NSURLConnection` class supports three ways of retrieving the content of a URL: synchronously, asynchronously using a completion handler block, and asynchronously using a custom delegate object.

To retrieve the contents of a URL synchronously: In code that runs *exclusively* on a background thread, you can call `sendSynchronousRequest:returningResponse:error:` to perform an HTTP request. This call returns when the request completes or an error occurs. For more details, see Retrieving Data Synchronously (page 38).

To retrieve the contents of a URL using a completion handler block: If you do not need to monitor the status of a request, but merely need to perform some operation when the data has been fully received, you can call `sendAsynchronousRequest:queue:completionHandler:`, passing a block to handle the results. For more details, see Retrieving Data Using a Completion Handler Block (page 38).

To retrieve the contents of a URL using a delegate object: Create a delegate class that implements at least the following delegate methods: `connection:didReceiveResponse:`, `connection:didReceiveData:`, `connection:didFailWithError:`, and `connectionDidFinishLoading:`. The supported delegate methods are defined in the `NSURLConnectionDelegate`, `NSURLConnectionDownloadDelegate`, and `NSURLConnectionDataDelegate` protocols.

The example in Listing 2-1 initiates a connection for a URL. This snippet begins by creating an `NSURLRequest` instance for the URL, specifying the cache access policy and the timeout interval for the connection. It then creates an `NSURLConnection` instance, specifying the request and a delegate. If `NSURLConnection` can't create a connection for the request, `initWithRequest:delegate:` returns `nil`. The snippet also creates an instance of `NSMutableData` to store the data that is incrementally provided to the delegate.

Listing 2-1 Creating a connection using `NSURLConnection`

```
// Create the request.
NSURLRequest *theRequest=[NSURLRequest requestWithURL:[NSURL
URLWithString:@"http://www.apple.com/"]
                        cachePolicy:NSURLRequestUseProtocolCachePolicy
                    timeoutInterval:60.0];

// Create the NSMutableData to hold the received data.
// receivedData is an instance variable declared elsewhere.
receivedData = [NSMutableData dataWithCapacity: 0];

// create the connection with the request
// and start loading the data
NSURLConnection *theConnection=[[NSURLConnection alloc] initWithRequest:theRequest
 delegate:self];
if (!theConnection) {
    // Release the receivedData object.
    receivedData = nil;

    // Inform the user that the connection failed.
}
```

The transfer starts immediately upon receiving the `initWithRequest:delegate:` message. It can be canceled any time before the delegate receives a `connectionDidFinishLoading:` or `connection:didFailWithError:` message by sending the connection a `cancel` message.

When the server has provided sufficient data to create an `NSURLResponse` object, the delegate receives a `connection:didReceiveResponse:` message. The delegate method can examine the provided `NSURLResponse` object and determine the expected content length of the data, MIME type, suggested filename, and other metadata provided by the server.

You should be prepared for your delegate to receive the `connection:didReceiveResponse:` message multiple times for a single connection; this can happen if the response is in multipart MIME encoding. Each time the delegate receives the `connection:didReceiveResponse:` message, it should reset any progress indication and discard all previously received data (except in the case of multipart responses). The example implementation in Listing 2-2 simply resets the length of the received data to 0 each time it is called.

Listing 2-2 Example `connection:didReceiveResponse:` implementation

```
- (void)connection:(NSURLConnection *)connection didReceiveResponse:(NSURLResponse
 *)response
{
    // This method is called when the server has determined that it
    // has enough information to create the NSURLResponse object.

    // It can be called multiple times, for example in the case of a
    // redirect, so each time we reset the data.

    // receivedData is an instance variable declared elsewhere.
    [receivedData setLength:0];
}
```

The delegate is periodically sent `connection:didReceiveData:` messages as the data is received. The delegate implementation is responsible for storing the newly received data. In the example implementation in Listing 2-3, the new data is appended to the NSMutableData object created in Listing 2-1 (page 33).

Listing 2-3 Example `connection:didReceiveData:` implementation

```
- (void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)data
{
    // Append the new data to receivedData.
    // receivedData is an instance variable declared elsewhere.
    [receivedData appendData:data];
}
```

You can also use the `connection:didReceiveData:` method to provide an indication of the connection's progress to the user. To do this, you must first obtain the expected content length by calling the `expectedContentLength` method on the URL response object in your `connection:didReceiveResponse:` delegate method. If the server does not provide length information, `expectedContentLength` returns `NSURLResponseUnknownLength`.

If an error occurs during the transfer, the delegate receives a `connection:didFailWithError:` message. The `NSError` object passed as the parameter specifies the details of the error. It also provides the URL of the request that failed in the user info dictionary using the key `NSURLErrorFailingURLStringErrorKey`.

After the delegate receives a `connection:didFailWithError:` message, it receives no further delegate messages for the specified connection.

The example in Listing 2-4 releases the connection, as well as any received data, and logs the error.

Listing 2-4 Example `connection:didFailWithError:` implementation

```
- (void)connection:(NSURLConnection *)connection
  didFailWithError:(NSError *)error
{
    // Release the connection and the data object
    // by setting the properties (declared elsewhere)
    // to nil.  Note that a real-world app usually
    // requires the delegate to manage more than one
    // connection at a time, so these lines would
    // typically be replaced by code to iterate through
    // whatever data structures you are using.
    theConnection = nil;
    receivedData = nil;

    // inform the user
    NSLog(@"Connection failed! Error - %@ %@",
          [error localizedDescription],
          [[error userInfo] objectForKey:NSURLErrorFailingURLStringErrorKey]);
}
```

Finally, if the connection succeeds in retrieving the request, the delegate receives the `connectionDidFinishLoading:` message. The delegate receives no further messages for the connection, and the app can release the `NSURLConnection` object.

The example implementation in Listing 2-5 logs the length of the received data and releases both the connection object and the received data.

Listing 2-5 Example `connectionDidFinishLoading:` implementation

```
- (void)connectionDidFinishLoading:(NSURLConnection *)connection
{
    // do something with the data
```

```
    // receivedData is declared as a property elsewhere
    NSLog(@"Succeeded! Received %d bytes of data",[receivedData length]);

    // Release the connection and the data object
    // by setting the properties (declared elsewhere)
    // to nil.  Note that a real-world app usually
    // requires the delegate to manage more than one
    // connection at a time, so these lines would
    // typically be replaced by code to iterate through
    // whatever data structures you are using.
    theConnection = nil;
    receivedData = nil;
}
```

This example represents the simplest implementation of a client using `NSURLConnection`. Additional delegate methods provide the ability to customize the handling of server redirects, authorization requests, and response caching.

Making a POST Request

You can make an HTTP or HTTPS POST request in nearly the same way you would make any other URL request (described in An Authentication Example (page 56)). The main difference is that you must first configure the `NSMutableURLRequest` object you provide to the `initWithRequest:delegate:` method.

You also need to construct the body data. You can do this in one of three ways:

- For uploading short, in-memory data, you should URL-encode an existing piece of data, as described in Continuing Without Credentials (page 55).
- For uploading file data from disk, call the `setHTTPBodyStream:` method to tell `NSMutableURLRequest` to read from an `NSInputStream` and use the resulting data as the body content.
- For large blocks of constructed data, call `CFStreamCreateBoundPair` to create a pair of streams, then call the `setHTTPBodyStream:` method to tell `NSMutableURLRequest` to use one of those streams as the source for its body content. By writing into the other stream, you can send the data a piece at a time.

  Depending on how you handle things on the server side, you may also want to URL-encode the data you send. (For details, see Continuing Without Credentials (page 55).)

If you are uploading data to a compatible server, the URL loading system also supports the `100` (Continue) HTTP status code, which allows an upload to continue where it left off in the event of an authentication error or other failure. To enable support for upload continuation, set the `Expect:` header on the request object to `100-continue`.

Listing 6-1 shows how to configure an `NSMutableURLRequest` object for a POST request.

Listing 2-6 Configuring an `NSMutableRequest` object for a POST request

```
// In body data for the 'application/x-www-form-urlencoded' content type,
// form fields are separated by an ampersand. Note the absence of a
// leading ampersand.
NSString *bodyData = @"name=Jane+Doe&address=123+Main+St";

NSMutableURLRequest *postRequest = [NSMutableURLRequest requestWithURL:[NSURL
URLWithString:@"https://www.apple.com"]];

// Set the request's content type to application/x-www-form-urlencoded
[postRequest setValue:@"application/x-www-form-urlencoded"
forHTTPHeaderField:@"Content-Type"];

// Designate the request a POST request and specify its body data
[postRequest setHTTPMethod:@"POST"];
[postRequest setHTTPBody:[NSData dataWithBytes:[bodyData UTF8String]
length:strlen([bodyData UTF8String])]];

// Initialize the NSURLConnection and proceed as described in
// Retrieving the Contents of a URL
```

To specify a different content type for the request, use the `setValue:forHTTPHeaderField:` method. If you do, make sure your body data is properly formatted for that content type.

To obtain a progress estimate for a POST request, implement a `connection:didSendBodyData:totalBytesWritten:totalBytesExpectedToWrite:` method in the connection's delegate. Note that this is not an exact measurement of upload progress, because the connection may fail or the connection may encounter an authentication challenge.

Retrieving Data Using a Completion Handler Block

The `NSURLConnection` class provides support for retrieving the contents of a resource represented by an `NSURLRequest` object in a asynchronous manner and calling a block when results are returned or when an error or timeout occurs. To do this, call the class method `sendAsynchronousRequest:queue:completionHandler:`, providing the request object, a completion handler block, and an `NSOperation` queue on which that block should run. When the request completes or an error occurs, the URL loading system calls that block with the result data or error information.

If the request succeeds, the contents of the request are passed to the callback handler block as an `NSData` object and an `NSURLResponse` object for the request. If `NSURLConnection` is unable to retrieve the URL, an `NSError` object is passed as the third parameter.

Note: This method has two significant limitations:

- Minimal support is provided for requests that require authentication. If the request requires authentication to make the connection, valid credentials must already be available in the `NSURLCredentialStorage` object or must be provided as part of the requested URL. If the credentials are not available or fail to authenticate, the URL loading system responds by sending the `NSURLProtocol` subclass handling the connection a `continueWithoutCredentialForAuthenticationChallenge:` message.
- There is no means of modifying the default behavior of response caching or accepting server redirects. When a connection attempt encounters a server redirect, the redirect is always honored. Likewise, the response data is stored in the cache according to the default support provided by the protocol implementation.

The `NSURLSession` class provides similar functionality without these limitations. For more information, read Using NSURLSession (page 16).

Retrieving Data Synchronously

The `NSURLConnection` class provides support for retrieving the contents of a resource represented by an `NSURLRequest` object in a synchronous manner using the class method `sendSynchronousRequest:returningResponse:error:`. Using this method is not recommended, because it has severe limitations:

- Unless you are writing a command-line tool, you must add additional code to ensure that the request does not run on your app's main thread.
- Minimal support is provided for requests that require authentication.

- There is no means of modifying the default behavior of response caching or accepting server redirects.

Important: If you retrieve data synchronously, you *must* ensure that the code in question can never run on your app's main thread. Network operations can take an arbitrarily long time to complete. If you attempt to perform those network operations synchronously on the main thread, the operations would block your app's execution until the data has been completely received, an error occurs, or the request times out. This causes a poor user experience, and can cause iOS to terminate your app.

If the request succeeds, the contents of the request are returned as an `NSData` object and an `NSURLResponse` object for the request is returned by reference. If `NSURLConnection` is unable to retrieve the URL, the method returns `nil` and any available `NSError` instance by reference in the appropriate parameter.

If the request requires authentication to make the connection, valid credentials must already be available in the `NSURLCredentialStorage` object or must be provided as part of the requested URL. If the credentials are not available or fail to authenticate, the URL loading system responds by sending the `NSURLProtocol` subclass handling the connection a `continueWithoutCredentialForAuthenticationChallenge:` message.

When a synchronous connection attempt encounters a server redirect, the redirect is always honored. Likewise, the response data is stored in the cache according to the default support provided by the protocol implementation.

Using NSURLDownload

Objective-CSwift

In OS X, `NSURLDownload` gives an application the ability to download the contents of a URL directly to disk. It provides an interface similar to `NSURLConnection`, adding an additional method for specifying the destination of the file. `NSURLDownload` can also decode commonly used encoding schemes such as MacBinary, BinHex, and gzip. Unlike `NSURLConnection`, data downloaded using `NSURLDownload` is not stored in the cache system.

If your application is not restricted to using Foundation classes, the WebKit framework includes WebDownload, a subclass of `NSURLDownload` that provides a user interface for authentication.

> **iOS Note:** The `NSURLDownload` class is not available in iOS, because downloading directly to the file system is discouraged. Use the `NSURLSession` or `NSURLConnection` class instead. See Using NSURLSession (page 16) and Using NSURLConnection (page 32) for more information.

Downloading to a Predetermined Destination

One usage pattern for `NSURLDownload` is downloading a file to a predetermined filename on disk. If the application knows the destination of the download, it can set it explicitly using `setDestination:allowOverwrite:`. Multiple `setDestination:allowOverwrite:` messages to an `NSURLDownload` instance are ignored.

The download starts immediately upon receiving the `initWithRequest:delegate:` message. It can be canceled any time before the delegate receives a `downloadDidFinish:` or `download:didFailWithError:` message by sending the download a `cancel` message.

The example in Listing 3-1 sets the destination, and thus requires the delegate only implement the `download:didFailWithError:` and `downloadDidFinish:` methods.

Listing 3-1 Using `NSURLDownload` with a predetermined destination file location

```
- (void)startDownloadingURL:sender
{
    // Create the request.
```

```
    NSURLRequest *theRequest = [NSURLRequest requestWithURL:[NSURL
URLWithString:@"http://www.apple.com"]

cachePolicy:NSURLRequestUseProtocolCachePolicy
                                        timeoutInterval:60.0];

    // Create the connection with the request and start loading the data.
NSURLDownload  *theDownload = [[NSURLDownload alloc] initWithRequest:theRequest
                                            delegate:self];
    if (theDownload) {
        // Set the destination file.
        [theDownload setDestination:@"/tmp" allowOverwrite:YES];
    } else {
        // inform the user that the download failed.
    }
}



- (void)download:(NSURLDownload *)download didFailWithError:(NSError *)error
{
    // Dispose of any references to the download object
    // that your app might keep.
    ...

    // Inform the user.
    NSLog(@"Download failed! Error - %@ %@",
          [error localizedDescription],
          [[error userInfo] objectForKey:NSURLErrorFailingURLStringErrorKey]);
}

- (void)downloadDidFinish:(NSURLDownload *)download
{
    // Dispose of any references to the download object
    // that your app might keep.
    ...
```

```
    // Do something with the data.
    NSLog(@"%@",@"downloadDidFinish");
}
```

The delegate can implement additional methods to customize the handling of authentication, server redirects, and file decoding.

Downloading a File Using the Suggested Filename

Sometimes the application must derive the destination filename from the downloaded data itself. This requires you to implement the delegate method `download:decideDestinationWithSuggestedFilename:` and call `setDestination:allowOverwrite:` with the suggested filename. The example in Listing 3-2 saves the downloaded file to the desktop using the suggested filename.

Listing 3-2 Using `NSURLDownload` with a filename derived from the download

```
- (void)startDownloadingURL:sender
{
    // Create the request.
    NSURLRequest *theRequest = [NSURLRequest requestWithURL:[NSURL
URLWithString:@"http://www.apple.com/index.html"]

cachePolicy:NSURLRequestUseProtocolCachePolicy
                                            timeoutInterval:60.0];


    // Create the download with the request and start loading the data.
NSURLDownload  *theDownload = [[NSURLDownload alloc] initWithRequest:theRequest
delegate:self];
    if (!theDownload) {
        // Inform the user that the download failed.
    }
}


- (void)download:(NSURLDownload *)download
decideDestinationWithSuggestedFilename:(NSString *)filename
{
```

```
    NSString *destinationFilename;
    NSString *homeDirectory = NSHomeDirectory();

    destinationFilename = [[homeDirectory stringByAppendingPathComponent:@"Desktop"]
        stringByAppendingPathComponent:filename];
    [download setDestination:destinationFilename allowOverwrite:NO];
}


- (void)download:(NSURLDownload *)download didFailWithError:(NSError *)error
{
    // Dispose of any references to the download object
    // that your app might keep.
    ...

    // Inform the user.
    NSLog(@"Download failed! Error - %@ %@",
          [error localizedDescription],
          [[error userInfo] objectForKey:NSURLErrorFailingURLStringErrorKey]);
}

- (void)downloadDidFinish:(NSURLDownload *)download
{
    // Dispose of any references to the download object
    // that your app might keep.
    ...

    // Do something with the data.
    NSLog(@"%@",@"downloadDidFinish");
}
```

The downloaded file is stored on the user's desktop with the name `index.html`, which was derived from the downloaded content. Passing `NO` to `setDestination:allowOverwrite:` prevents an existing file from being overwritten by the download. Instead, a unique filename is created by inserting a sequential number after the filename—for example, `index-1.html`.

The delegate is informed when a file is created on disk if it implements the `download:didCreateDestination:` method. This method also gives the application the opportunity to determine the finalized filename with which the download is saved.

The example in Listing 3-3 logs the finalized filename.

Listing 3-3 Logging the finalized filename using `download:didCreateDestination:`

```
-(void)download:(NSURLDownload *)download didCreateDestination:(NSString *)path
{
    // path now contains the destination path
    // of the download, taking into account any
    // unique naming caused by -setDestination:allowOverwrite:
    NSLog(@"Final file destination: %@",path);
}
```

This message is sent to the delegate after it has been given an opportunity to respond to the `download:shouldDecodeSourceDataOfMIMEType:` and `download:decideDestinationWithSuggestedFilename:` messages.

Displaying Download Progress

You can determine the progress of a download by implementing the delegate methods `download:didReceiveResponse:` and `download:didReceiveDataOfLength:`.

The `download:didReceiveResponse:` method provides the delegate an opportunity to determine the expected content length from the `NSURLResponse`. The delegate should reset the progress each time it receives this message.

The example implementation in Listing 3-4 demonstrates using these methods to provide progress feedback to the user.

Listing 3-4 Displaying the download progress

```
- (void)setDownloadResponse:(NSURLResponse *)aDownloadResponse
{
    // downloadResponse is an instance variable defined elsewhere.
    downloadResponse = aDownloadResponse;
```

```
}

- (void)download:(NSURLDownload *)download didReceiveResponse:(NSURLResponse
*)response
{
    // Reset the progress, this might be called multiple times.
    // bytesReceived is an instance variable defined elsewhere.
    bytesReceived = 0;

    // Store the response to use later.
    [self setDownloadResponse:response];
}

- (void)download:(NSURLDownload *)download didReceiveDataOfLength:(unsigned)length
{
    long long expectedLength = [[self downloadResponse] expectedContentLength];

    bytesReceived = bytesReceived + length;

    if (expectedLength != NSURLResponseUnknownLength) {
        // If the expected content length is
        // available, display percent complete.
        float percentComplete = (bytesReceived/(float)expectedLength)*100.0;
        NSLog(@"Percent complete - %f",percentComplete);
    } else {
        // If the expected content length is
        // unknown, just log the progress.
        NSLog(@"Bytes received - %d",bytesReceived);
    }
}
```

The delegate receives a `download:didReceiveResponse:` message before it begins receiving `download:didReceiveDataOfLength:` messages.

Resuming Downloads

In some cases, you can resume a download that was canceled or that failed while in progress. To do so, first make sure your original download doesn't delete its data upon failure by passing `NO` to the download's `setDeletesFileUponFailure:` method. If the original download fails, you can obtain its data with the `resumeData` method. You can then initialize a new download with the `initWithResumeData:delegate:path:` method. When the download resumes, the download's delegate receives the `download:willResumeWithResponse:fromByte:` message.

You can resume a download only if both the protocol of the connection and the MIME type of the file being downloaded support resuming. You can determine whether your file's MIME type is supported with the `canResumeDownloadDecodedWithEncodingMIMEType:` method.

Decoding Encoded Files

`NSURLDownload` provides support for decoding the MacBinary, BinHex, and gzip file formats. If `NSURLDownload` determines that a file is encoded in a supported format, it attempts to send the delegate a `download:shouldDecodeSourceDataOfMIMEType:` message. If the delegate implements this method, it should examine the passed MIME type and return `YES` if the file should be decoded.

The example in Listing 3-5 compares the MIME type of the file and allows decoding of MacBinary and BinHex encoded content.

Listing 3-5 Example implementation of `download:shouldDecodeSourceDataOfMIMEType:` method

```
- (BOOL)download:(NSURLDownload *)download
     shouldDecodeSourceDataOfMIMEType:(NSString *)encodingType
{
    BOOL shouldDecode = NO;

    if ([encodingType isEqual:@"application/macbinary"]) {
        shouldDecode = YES;
    } else if ([encodingType isEqual:@"application/binhex"]) {
        shouldDecode = YES;
    } else if ([encodingType isEqual:@"application/x-gzip"]) {
        shouldDecode = NO;
    }
    return shouldDecode;
```

```
}
```

Encoding URL Data

To URL-encode strings, use the Core Foundation functions `CFURLCreateStringByAddingPercentEscapes` and `CFURLCreateStringByReplacingPercentEscapesUsingEncoding`. These functions allow you to specify a list of characters to encode in addition to high-ASCII (`0x80–0xff`) and nonprintable characters.

According to RFC 3986, the reserved characters in a URL are:

```
      reserved    = gen-delims / sub-delims

      gen-delims  = ":" / "/" / "?" / "#" / "[" / "]" / "@"

      sub-delims  = "!" / "$" / "&" / "'" / "(" / ")"
                  / "*" / "+" / "," / ";" / "="
```

Therefore, to properly URL-encode a UTF-8 string for inclusion in a URL, you should do the following:

```
CFStringRef originalString = ...

CFStringRef encodedString = CFURLCreateStringByAddingPercentEscapes(
    kCFAllocatorDefault,
    originalString,
    NULL,
    CFSTR(":/?#[]@!$&'()*+,;="),
    kCFStringEncodingUTF8);
```

If you want to decode a URL fragment, you must first split the URL string into its constituent parts (fields and path parts). If you do not decode it, you will be unable to tell the difference (for example) between an encoded ampersand that was originally part of the contents of a field and a bare ampersand that indicated the end of the field.

After you have broken the URL into parts, you can decode each part as follows:

```
CFStringRef decodedString = CFURLCreateStringByReplacingPercentEscapesUsingEncoding(
    kCFAllocatorDefault,
    encodedString,
    CFSTR(""),
    kCFStringEncodingUTF8);
```

Important: Although the `NSString` class provides built-in methods for adding percent escapes, you usually should *not* use them. These methods assume you are passing them a string containing a series of ampersand-separated values, and as a result, you cannot use them to properly URL-encode any string that contains an ampersand. If you try to use them for something that does, your code could be vulnerable to a URL string injection attack (security hole), depending on how the software at the other end handles the malformed URL.

Handling Redirects and Other Request Changes

A redirect occurs when a server responds to a request by indicating that the client should make a new request to a different URL. The `NSURLSession`, `NSURLConnection`, and `NSURLDownload` classes notify their delegates when this occurs.

To handle a redirect, your URL loading class delegate must implement one of the following delegate methods:

- For `NSURLSession`, implement the `URLSession:task:willPerformHTTPRedirection:newRequest:completionHandler:` delegate method.
- For `NSURLConnection`, implement the `connection:willSendRequest:redirectResponse:` delegate method.
- For `NSURLDownload`, implement the `download:willSendRequest:redirectResponse:` delegate method.

In these methods, the delegate can examine the new request and the response that caused the redirect, and can return a new request object through the completion handler for `NSURLSession` or through the return value for `NSURLConnection` and `NSURLDownload`.

The delegate can do any of the following:

- Allow the redirect by simply returning the provided request.
- Create a new request, pointing to a different URL, and return that request.
- Reject the redirect and receive any existing data from the connection by returning `nil`.

In addition, the delegate can cancel both the redirect and the connection. With `NSURLSession`, the delegate does this by sending the `cancel` message to the task object. With the `NSURLConnection` or `NSURLDownload` APIs, the delegate does this by sending the `cancel` message to the `NSURLConnection` or `NSURLDownload` object.

The delegate also receives the `connection:willSendRequest:redirectResponse:` message if the `NSURLProtocol` subclass that handles the request has changed the `NSURLRequest` in order to standardize its format, for example, changing a request for `http://www.apple.com` to `http://www.apple.com/`. This

occurs because the standardized, or canonical, version of the request is used for cache management. In this special case, the response passed to the delegate is `nil` and the delegate should simply return the provided request.

The example implementation in Listing 5-1 allows canonical changes and denies all server redirects.

Listing 5-1 Example of an implementation of `connection:willSendRequest:redirectResponse:`

```
#if FOR_NSURLSESSION
- (void)URLSession:(NSURLSession *)session
        task:(NSURLSessionTask *)task
        willPerformHTTPRedirection:(NSHTTPURLResponse *)redirectResponse
        newRequest:(NSURLRequest *)request
        completionHandler:(void (^)(NSURLRequest *))completionHandler
#elif FOR_NSURLCONNECTION
-(NSURLRequest *)connection:(NSURLConnection *)connection
            willSendRequest:(NSURLRequest *)request
           redirectResponse:(NSURLResponse *)redirectResponse
#else // FOR_NSURLDOWNLOAD
-(NSURLRequest *)download:(NSURLConnection *)connection
            willSendRequest:(NSURLRequest *)request
           redirectResponse:(NSURLResponse *)redirectResponse
#endif
{
    NSURLRequest *newRequest = request;
    if (redirectResponse) {
        newRequest = nil;
    }

#if FOR_NSURLSESSION
    completionHandler(newRequest);
#else
    return newRequest;
#endif
}
```

If the delegate doesn't provide an implementation for an appropriate redirect handling delegate method, all canonical changes and server redirects are allowed.

Authentication Challenges and TLS Chain Validation

Objective-CSwift

An `NSURLRequest` object often encounters an **authentication challenge**, or a request for credentials from the server it is connecting to. The `NSURLSession`, `NSURLConnection`, and `NSURLDownload` classes notify their delegates when a request encounters an authentication challenge, so that they can act accordingly.

> **Important:** The URL loading system classes do not call their delegates to handle request challenges unless the server response contains a `WWW-Authenticate` header. Other authentication types, such as proxy authentication and TLS trust validation do not require this header.

Deciding How to Respond to an Authentication Challenge

If an `NSURLRequest` object requires authentication, the way that the challenge is presented to your app varies depending on whether the request is performed by an `NSURLSession` object, an `NSURLConnection` object, or an `NSURLDownload` object:

- If the request is associated with an `NSURLSession` object, all authentication requests are passed to the delegate, regardless of authentication type.
- If the request is associated with an `NSURLConnection` or `NSURLDownload` object, that object's delegate receives a `connection:canAuthenticateAgainstProtectionSpace:` (or `download:canAuthenticateAgainstProtectionSpace:`) message. This allows the delegate to analyze properties of the server, including its protocol and authentication method, before attempting to authenticate against it. If your delegate is not prepared to authenticate against the server's protection space, you can return `NO`, and the system attempts to authenticate with information from the user's keychain.
- If the delegate of an `NSURLConnection` or `NSURLDownload` object does not implement the `connection:canAuthenticateAgainstProtectionSpace:` (or `download:canAuthenticateAgainstProtectionSpace:`) method and the protection space uses client certificate authentication or server trust authentication, the system behaves as if you had returned `NO`. The system behaves as if you returned `YES` for all other authentication types.

Next, if your delegate agrees to handle authentication and there are no valid credentials available, either as part of the requested URL or in the shared `NSURLCredentialStorage`, the delegate receives one of the following messages:

```
URLSession:didReceiveChallenge:completionHandler:
URLSession:task:didReceiveChallenge:completionHandler:
connection:didReceiveAuthenticationChallenge:
download:didReceiveAuthenticationChallenge:
```

In order for the connection to continue, the delegate has three options:

- Provide authentication credentials.
- Attempt to continue without credentials.
- Cancel the authentication challenge.

To help determine the correct course of action, the `NSURLAuthenticationChallenge` instance passed to the method contains information about what triggered the authentication challenge, how many attempts were made for the challenge, any previously attempted credentials, the `NSURLProtectionSpace` that requires the credentials, and the sender of the challenge.

If the authentication challenge has tried to authenticate previously and failed (for example, if the user changed his or her password on the server), you can obtain the attempted credentials by calling `proposedCredential` on the authentication challenge. The delegate can then use these credentials to populate a dialog that it presents to the user.

Calling `previousFailureCount` on the authentication challenge returns the total number of previous authentication attempts, including those from different authentication protocols. The delegate can provide this information to the user, to determine whether the credentials it supplied previously are failing, or to limit the maximum number of authentication attempts.

Responding to an Authentication Challenge

The following are the three ways you can respond to the `connection:didReceiveAuthenticationChallenge:` delegate method.

Providing Credentials

To attempt to authenticate, the application should create an `NSURLCredential` object with authentication information of the form expected by the server. You can determine the server's authentication method by calling `authenticationMethod` on the protection space of the provided authentication challenge. Some authentication methods supported by `NSURLCredential` are:

- HTTP basic authentication (`NSURLAuthenticationMethodHTTPBasic`) requires a user name and password. Prompt the user for the necessary information and create an `NSURLCredential` object with `credentialWithUser:password:persistence:`.
- HTTP digest authentication (`NSURLAuthenticationMethodHTTPDigest`), like basic authentication, requires a user name and password. (The digest is generated automatically.) Prompt the user for the necessary information and create an `NSURLCredential` object with `credentialWithUser:password:persistence:`.
- Client certificate authentication (`NSURLAuthenticationMethodClientCertificate`) requires the system identity and all certificates needed to authenticate with the server. Create an `NSURLCredential` object with `credentialWithIdentity:certificates:persistence:`.
- Server trust authentication (`NSURLAuthenticationMethodServerTrust`) requires a trust provided by the protection space of the authentication challenge. Create an `NSURLCredential` object with `credentialForTrust:`.

After you've created the `NSURLCredential` object:

- For `NSURLSession`, pass the object to the authentication challenge's sender using the provided completion handler block.
- For `NSURLConnection` and `NSURLDownload`, pass the object to the authentication challenge's sender with `useCredential:forAuthenticationChallenge:`.

Continuing Without Credentials

If the delegate chooses not to provide a credential for the authentication challenge, it can attempt to continue without one.

- For `NSURLSession`, pass one of the following values to the provided completion handler block:

  `NSURLSessionAuthChallengePerformDefaultHandling` processes the request as though the delegate did not provide a delegate method to handle the challenge.

  `NSURLSessionAuthChallengeRejectProtectionSpace` rejects the challenge. Depending on the authentication types allowed by the server's response, the URL loading class may call this delegate method more than once, for additional protection spaces.

- For `NSURLConnection` and `NSURLDownload`, call `continueWithoutCredentialsForAuthenticationChallenge:` on `[challenge sender]`.

  Depending on the protocol implementation, continuing without credentials may either cause the connection to fail, resulting in a `connectionDidFailWithError:` message, or return alternate URL contents that don't require authentication.

Canceling the Connection

The delegate may also choose to cancel the authentication challenge.

- For `NSURLSession`, pass `NSURLSessionAuthChallengeCancelAuthenticationChallenge` to the provided completion handler block.
- For `NSURLConnection` or `NSURLDownload`, call `cancelAuthenticationChallenge:` on `[challenge sender]`. The delegate receives a `connection:didCancelAuthenticationChallenge:` message, providing the opportunity to give the user feedback.

An Authentication Example

The implementation shown in Listing 6-1 responds to the challenge by creating an `NSURLCredential` instance with a user name and password supplied by the application's preferences. If the authentication has failed previously, it cancels the authentication challenge and informs the user.

Listing 6-1 An example of using the `connection:didReceiveAuthenticationChallenge:` delegate method

```
-(void)connection:(NSURLConnection *)connection
       didReceiveAuthenticationChallenge:(NSURLAuthenticationChallenge *)challenge
{
    if ([challenge previousFailureCount] == 0) {
        NSURLCredential *newCredential;
        newCredential = [NSURLCredential credentialWithUser:[self preferencesName]
                                                   password:[self preferencesPassword]

persistence:NSURLCredentialPersistenceNone];
        [[challenge sender] useCredential:newCredential
               forAuthenticationChallenge:challenge];
    } else {
        [[challenge sender] cancelAuthenticationChallenge:challenge];
        // inform the user that the user name and password
```

```
        // in the preferences are incorrect
        [self showPreferencesCredentialsAreIncorrectPanel:self];
    }
}
```

If the delegate doesn't implement `connection:didReceiveAuthenticationChallenge:` and the request requires authentication, valid credentials must already be available in the URL credential storage or must be provided as part of the requested URL. If the credentials are not available or if they fail to authenticate, a `continueWithoutCredentialForAuthenticationChallenge:` message is sent by the underlying implementation.

Performing Custom TLS Chain Validation

In the NSURL family of APIs, TLS chain validation is handled by your app's authentication delegate method, but instead of providing credentials to authenticate the user (or your app) to the server, your app instead checks the credentials that the server provides during the TLS handshake, then tells the URL loading system whether it should accept or reject those credentials.

If you need to perform chain validation in a nonstandard way (such as accepting a specific self-signed certificate for testing), your app must do the following:

- For `NSURLSession`, implement either the `URLSession:didReceiveChallenge:completionHandler:` or `URLSession:task:didReceiveChallenge:completionHandler:` delegate method. If you implement both, the session-level method is responsible for handling the authentication.
- For `NSURLConnection` and `NSURLDownload`, implement the `connection:canAuthenticateAgainstProtectionSpace:` or `download:canAuthenticateAgainstProtectionSpace:` method and return `YES` if the protection space has an authentication type of `NSURLAuthenticationMethodServerTrust`.

  Then, implement the `connection:didReceiveAuthenticationChallenge:` or `download:didReceiveAuthenticationChallenge:` method to handle the authentication.

Within your authentication handler delegate method, you should check to see if the challenge protection space has an authentication type of `NSURLAuthenticationMethodServerTrust`, and if so, obtain the `serverTrust` information from that protection space.

For additional details and code snippets (based on `NSURLConnection`), read Overriding TLS Chain Validation Correctly.

Understanding Cache Access

The URL loading system provides a composite on-disk and in-memory cache of responses to requests. This cache allows an application to reduce its dependency on a network connection and increase its performance.

Using the Cache for a Request

An `NSURLRequest` instance specifies how the local cache is used by setting the cache policy to one of the `NSURLRequestCachePolicy` values: `NSURLRequestUseProtocolCachePolicy`, `NSURLRequestReloadIgnoringCacheData`, `NSURLRequestReturnCacheDataElseLoad`, or `NSURLRequestReturnCacheDataDontLoad`.

The default cache policy for an `NSURLRequest` instance is `NSURLRequestUseProtocolCachePolicy`. The `NSURLRequestUseProtocolCachePolicy` behavior is protocol specific and is defined as being the best conforming policy for the protocol.

Setting the cache policy to `NSURLRequestReloadIgnoringCacheData` causes the URL loading system to load the data from the originating source, ignoring the cache completely.

The `NSURLRequestReturnCacheDataElseLoad` cache policy causes the URL loading system to use cached data, ignoring its age or expiration date, and to load the data from the originating source only if there is no cached version.

The `NSURLRequestReturnCacheDataDontLoad` policy allows an application to specify that only data in the cache should be returned. Attempting to create an `NSURLConnection` or `NSURLDownload` instance with this cache policy returns `nil` immediately if the response is not in the local cache. This is similar in function to an "offline" mode and never brings up a network connection.

> **Note:** Currently, only responses to HTTP and HTTPS requests are cached. The FTP and `file` protocols attempt to access the originating source as allowed by the cache policy. Custom `NSURLProtocol` classes can optionally provide caching.

Cache Use Semantics for the HTTP Protocol

The most complicated cache use situation is when a request uses the HTTP protocol and has set the cache policy to `NSURLRequestUseProtocolCachePolicy`.

If an `NSCachedURLResponse` does not exist for the request, then the URL loading system fetches the data from the originating source.

If a cached response exists for the request, the URL loading system checks the response to determine if it specifies that the contents must be revalidated.

If the contents must be revalidated, the URL loading system makes a HEAD request to the originating source to see if the resource has changed. If it has not changed, then the URL loading system returns the cached response. If it has changed, the URL loading system fetches the data from the originating source.

If the cached response doesn't specify that the contents must be revalidated, the URL loading system examines the maximum age or expiration specified in the cached response. If the cached response is recent enough, then the URL loading system returns the cached response. If the response is stale, the URL loading system makes a HEAD request to the originating source to determine whether the resource has changed. If so, the URL loading system fetches the resource from the originating source. Otherwise, it returned the cached response.

RFC 2616, Section 13 (http://www.w3.org/Protocols/rfc2616/rfc2616-sec13.html#sec13) specifies the semantics involved in detail.

Controlling Caching Programmatically

By default, the data for a connection is cached based on the request's cache policy, as interpreted by the `NSURLProtocol` subclass that handles the request.

If your app needs more precise programmatic control over caching (and if the protocol supports caching), you can implement a delegate method that allows your app to determine on a per-request basis whether a particular response should be cached.

- For `NSURLSession` data and upload tasks, implement the `URLSession:dataTask:willCacheResponse:completionHandler:` method.

This delegate method is called *only* for data and upload tasks. The caching policy for download tasks is determined by the specified cache policy exclusively.

- For `NSURLConnection`, implement the `connection:willCacheResponse:` method.

For `NSURLSession`, your delegate method calls a completion handler block to tell the session what to cache. For `NSURLConnection`, your delegate method returns the object that the connection should cache.

In either case, the delegate typically provides one the following values:

- The provided response object to allow caching
- A newly created response object to cache a modified response—for example, a response with a storage policy that allows caching to memory but not to disk
- `NULL` to prevent caching

Your delegate method can also insert objects into the `userInfo` dictionary associated with an `NSCachedURLResponse` object, causing those objects to be stored in the cache as part of the response.

Important: If you are using `NSURLSession` and you implement this delegate method, your delegate method *must always* call the provided completion handler. Otherwise, your app leaks memory.

The example in Listing 7-1 prevents the on-disk caching of HTTPS responses. It also adds the current date to the user info dictionary for responses that are cached.

Listing 7-1 Example `connection:withCacheResponse:` implementation

```
-(NSCachedURLResponse *)connection:(NSURLConnection *)connection
                  willCacheResponse:(NSCachedURLResponse *)cachedResponse
{
    NSCachedURLResponse *newCachedResponse = cachedResponse;

    NSDictionary *newUserInfo;
    newUserInfo = [NSDictionary dictionaryWithObject:[NSDate date]
                                              forKey:@"Cached Date"];
    if ([[[[cachedResponse response] URL] scheme] isEqual:@"https"]) {
#if ALLOW_IN_MEMORY_CACHING
        newCachedResponse = [[NSCachedURLResponse alloc]
                                 initWithResponse:[cachedResponse response]
                                     data:[cachedResponse data]
```

```
                                    userInfo:newUserInfo

storagePolicy:NSURLCacheStorageAllowedInMemoryOnly];
#else // !ALLOW_IN_MEMORY_CACHING
        newCachedResponse = nil
#endif // ALLOW_IN_MEMORY_CACHING
    } else {
        newCachedResponse = [[NSCachedURLResponse alloc]
                                initWithResponse:[cachedResponse response]
                                    data:[cachedResponse data]
                                    userInfo:newUserInfo
                                   storagePolicy:[cachedResponse storagePolicy]];
    }
    return newCachedResponse;
}
```

Cookies and Custom Protocols

SwiftObjective-C

If your app needs to manage cookies programmatically, such as adding and deleting cookies or determining which cookies should be accepted, read Cookie Storage (page 62).

If your app needs to support a URL-based protocol that `NSURL` does not support natively, you can register your own custom protocol class that provides the needed support. To learn more, read Protocol Support (page 63).

Cookie Storage

Due to the stateless nature of the HTTP protocol, clients often use cookies to provide persistent storage of data across URL requests. The URL loading system provides interfaces to create and manage cookies, to send cookies as part of an HTTP request, and to receive cookies when interpreting a web server's response.

The `NSHTTPCookie` class encapsulates a cookie, providing accessors for many of the common cookie attributes. This class also provides methods to convert HTTP cookie headers to `NSHTTPCookie` instances and convert an `NSHTTPCookie` instance to headers suitable for use with an `NSURLRequest` object. The URL loading system automatically sends any stored cookies appropriate for an `NSURLRequest` object unless the request specifies not to send cookies. Likewise, cookies returned in an `NSURLResponse` object are accepted in accordance with the current cookie acceptance policy.

The `NSHTTPCookieStorage` class provides the interface for managing the collection of `NSHTTPCookie` objects shared by all apps.

> **iOS Note:** Cookies are not shared between apps in iOS.

`NSHTTPCookieStorage` allows an app to specify a cookie acceptance policy. The cookie acceptance policy controls whether cookies should always be accepted, never be accepted, or be accepted only from the same domain as the main document URL.

> **Note:** Changing the cookie acceptance policy in an app affects the cookie acceptance policy for all other running apps.

When another app changes the cookie storage or the cookie acceptance policy, `NSHTTPCookieStorage` notifies an app by posting the `NSHTTPCookieManagerCookiesChangedNotification` and `NSHTTPCookieStorageAcceptPolicyChangedNotification` notifications.

For more information, see *NSHTTPCookieStorage Class Reference* and *NSHTTPCookie Class Reference*.

Protocol Support

The URL loading system design allows a client app to extend the protocols that are supported for transferring data. The URL loading system natively supports the `http`, `https`, `file`, `ftp`, and `data` protocols.

You can implement a custom protocol by subclassing `NSURLProtocol` and then registering the new class with the URL loading system using the `NSURLProtocol` class method `registerClass:`. When an `NSURLSession`, `NSURLConnection`, or `NSURLDownload` object initiates a connection for an `NSURLRequest` object, the URL loading system consults each of the registered classes in the reverse order of their registration. The first class that returns `YES` for a `canInitWithRequest:` message is used to handle the request.

If your custom protocol requires additional properties for its requests or responses, you support them by creating categories on the `NSURLRequest`, `NSMutableURLRequest`, and `NSURLResponse` classes that provide accessors for those properties. The `NSURLProtocol` class provides methods for setting and getting property values in those accessors.

The URL loading system is responsible for creating and releasing `NSURLProtocol` instances when connections start and complete. Your app should never create an instance of `NSURLProtocol` directly.

When an `NSURLProtocol` subclass is initialized by the URL loading system, it is provided a client object that conforms to the `NSURLProtocolClient` protocol. The `NSURLProtocol` subclass sends messages from the `NSURLProtocolClient` protocol to the client object to inform the URL loading system of its actions as it creates a response, receives data, redirects to a new URL, requires authentication, and completes the load. If the custom protocol supports authentication, then it must conform to the `NSURLAuthenticationChallengeSender` protocol.

For more information, see *NSURLProtocol Class Reference*.

Life Cycle of a URL Session

You can use the `NSURLSession` API in two ways: with a system-provided delegate or with your own delegate. In general, you must use your own delegate if your app does any of the following:

- Uses background sessions to download or upload content while your app is not running.
- Performs custom authentication.
- Performs custom SSL certificate verification.
- Decides whether a transfer should be downloaded to disk or displayed based on the MIME type returned by the server or other similar criteria.
- Uploads data from a body stream (as opposed to an `NSData` object).
- Limits caching programmatically.
- Limits HTTP redirects programmatically.

If your app does not need to do any of these things, your app can use the system-provided delegates. Depending on which technique you choose, you should read one of the following sections:

- Life Cycle of a URL Session with System-Provided Delegates (page 64) provides a lightweight view of how your code creates and uses a URL session. You should read this section even if you intend to write your own delegate, because it gives you a complete picture of what your code must do to configure the object and use it.
- Life Cycle of a URL Session with Custom Delegates (page 66) provides a complete view of every step in the operation of a URL session. You should refer to this section to help you understand how the session interacts with its delegate. In particular, this explains when each of the delegate methods is called.

Life Cycle of a URL Session with System-Provided Delegates

If you are using the `NSURLSession` class without providing a delegate object, the system-provided delegate handles many of the details for you. Here is the basic sequence of method calls that your app must make and completion handler calls that your app receives when using `NSURLSession` with the system-provided delegate:

1. Create a session configuration. For background sessions, this configuration must contain a unique identifier. Store that identifier, and use it to reassociate with the session if your app crashes or is terminated or suspended.

2. Create a session, specifying a configuration object and a `nil` delegate.
3. Create task objects within a session that each represent a resource request.

   Each task starts out in a suspended state. After your app calls `resume` on the task, it begins downloading the specified resource.

   The task objects are subclasses of `NSURLSessionTask`—`NSURLSessionDataTask`, `NSURLSessionUploadTask`, or `NSURLSessionDownloadTask`, depending on the behavior you are trying to achieve. These objects are analogous to `NSURLConnection` objects, but give you more control and a unified delegate model.

   Although your app can (and typically should) add more than one task to a session, for simplicity, the remaining steps describe the life cycle in terms of a single task.

   > **Important:** If you are using the `NSURLSession` class without providing delegates, your app must create tasks using a call that takes a `completionHandler` parameter, because otherwise it cannot obtain data from the class.

4. For a download task, during the transfer from the server, if the user tells your app to pause the download, cancel the task by calling `cancelByProducingResumeData:` method. Later, pass the returned resume data to either the `downloadTaskWithResumeData:` or `downloadTaskWithResumeData:completionHandler:` method to create a new download task that continues the download.
5. When a task completes, the `NSURLSession` object calls the task's completion handler.

   > **Note:** `NSURLSession` does not report server errors through the error parameter. The only errors your app receives through the error parameter are client-side errors, such as being unable to resolve the hostname or connect to the host. The error codes are described in `URL Loading System Error Codes`.
   >
   > Server-side errors are reported through the HTTP status code in the `NSHTTPURLResponse` object. For more information, read the documentation for the `NSHTTPURLResponse` and `NSURLResponse` classes.

6. When your app no longer needs a session, invalidate it by calling either `invalidateAndCancel` (to cancel outstanding tasks) or `finishTasksAndInvalidate` (to allow outstanding tasks to finish before invalidating the object).

Life Cycle of a URL Session with Custom Delegates

You can often use the `NSURLSession` API without providing a delegate. However, if you are using the `NSURLSession` API for background downloads and uploads, or if you need to handle authentication or caching in a nondefault manner, you must provide a delegate that conforms to the session delegate protocol, one or more task delegate protocols, or some combination of these protocols. This delegate serves many purposes:

- When used with download tasks, the `NSURLSession` object uses the delegate to provide your app with a file URL where it can obtain the downloaded data.

  Delegates are required for all background downloads and uploads. These delegates must provide all of the delegate methods in the `NSURLSessionDownloadDelegate` protocol.
- Delegates can handle certain authentication challenges.
- Delegates provide body streams for uploading stream-based data to the remote server.
- Delegates can decide whether to follow HTTP redirects or not.
- The `NSURLSession` object uses the delegate to provide your app with the status of each transfer. Data task delegates receive both an initial call, in which you can convert the request into a download, and subsequent calls, which provide pieces of data as they arrive from the remote server.
- Delegates are one way in which the `NSURLSession` object can tell your app when a transfer is complete.

If you are using custom delegates with a URL session (required for background tasks), the complete life cycle of a URL session is more complex. Here is the basic sequence of method calls that your app must make and delegate calls that your app receives when using `NSURLSession` with a custom delegate:

1. Create a session configuration. For background sessions, this configuration must contain a unique identifier. Store that identifier, and use it to reassociate with the session if your app crashes or is terminated or suspended.
2. Create a session, specifying a configuration object and, optionally, a delegate.
3. Create task objects within a session that each represent a resource request.

   Each task starts out in a suspended state. After your app calls `resume` on the task, it begins downloading the specified resource.

   The task objects are subclasses of `NSURLSessionTask`—`NSURLSessionDataTask`, `NSURLSessionUploadTask`, or `NSURLSessionDownloadTask`, depending on the behavior you are trying to achieve. These objects are analogous to `NSURLConnection` objects, but give you more control and a unified delegate model.

   Although your app can (and typically should) add more than one task to a session, for simplicity, the remaining steps describe the life cycle in terms of a single task.

4. 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.

    If authentication fails for an upload task, if the task's data is provided from a stream, the `NSURLSession` object calls the delegate's `URLSession:task:needNewBodyStream:` delegate method. The delegate must then provide a new `NSInputStream` object to provide the body data for the new request.

    For more information about writing an authentication delegate method for `NSURLSession`, read Authentication Challenges and TLS Chain Validation (page 53).
5. Upon receiving an HTTP redirect response, the `NSURLSession` object calls the delegate's `URLSession:task:willPerformHTTPRedirection:newRequest:completionHandler:` method. That delegate method calls the provided completion handler with either the provided `NSURLRequest` object (to follow the redirect), a new `NSURLRequest` object (to redirect to a different URL), or `nil` (to treat the redirect's response body as a valid response and return it as the result).
    - If the redirect is followed, go back to step 4 (authentication challenge handling).
    - If the delegate does not implement this method, the redirect is followed up to the maximum number of redirects.
6. For a (re-)download task created by calling `downloadTaskWithResumeData:` or `downloadTaskWithResumeData:completionHandler:`, `NSURLSession` calls the delegate's `URLSession:downloadTask:didResumeAtOffset:expectedTotalBytes:` method with the new task object.

7. For a data task, the `NSURLSession` object calls the delegate's `URLSession:dataTask:didReceiveResponse:completionHandler:` method. Decide whether to convert the data task into a download task, and then call the completion callback to continue receiving data or downloading data.

   If your app chooses to convert the data task to a download task, `NSURLSession` calls the delegate's `URLSession:dataTask:didBecomeDownloadTask:` method with the new download task as a parameter. After this call, the delegate receives no further callbacks from the data task, and begins receiving callbacks from the download task.

8. If the task was created with `uploadTaskWithStreamedRequest:`, `NSURLSession` calls the delegate's `URLSession:task:needNewBodyStream:` method to provide the body data.

9. During the initial upload of body content to the server (if applicable), the delegate periodically receives `URLSession:task:didSendBodyData:totalBytesSent:totalBytesExpectedToSend:` callbacks that report the progress of the upload.

10. During the transfer from the server, the task delegate periodically receives a callback to report the progress of the transfer. For a download task, the session calls the delegate's `URLSession:downloadTask:didWriteData:totalBytesWritten:totalBytesExpectedToWrite:` method with the number of bytes successfully written to disk. For a data task, the session calls the delegate's `URLSession:dataTask:didReceiveData:` method with the actual pieces of data as they are received.

    For a download task, during the transfer from the server, if the user tells your app to pause the download, cancel the task by calling the `cancelByProducingResumeData:` method.

    Later, if the user asks your app to resume the download, pass the returned resume data to either the `downloadTaskWithResumeData:` or `downloadTaskWithResumeData:completionHandler:` method to create a new download task that continues the download, then go to step 3 (creating and resuming task objects).

11. For a data task, the `NSURLSession` object calls the delegate's `URLSession:dataTask:willCacheResponse:completionHandler:` method. Your app should then decide whether to allow caching. If you do not implement this method, the default behavior is to use the caching policy specified in the session's configuration object.

12. If a download task completes successfully, then the `NSURLSession` object calls the task's `URLSession:downloadTask:didFinishDownloadingToURL:` method with the location of a *temporary* file. Your app must either read the response data from this file or move it to a permanent location in your app's sandbox container directory before this delegate method returns.

13. When any task completes, the `NSURLSession` object calls the delegate's `URLSession:task:didCompleteWithError:` method with either an error object or `nil` (if the task completed successfully).

If the task failed, most apps should retry the request until either the user cancels the download or the server returns an error indicating that the request will never succeed. Your app should not retry immediately, however. Instead, it should use reachability APIs to determine whether the server is reachable, and should make a new request only when it receives a notification that reachability has changed.

If the download task can be resumed, the `NSError` object's `userInfo` dictionary contains a value for the `NSURLSessionDownloadTaskResumeData` key. Your app should pass this value to call `downloadTaskWithResumeData:` or `downloadTaskWithResumeData:completionHandler:` to create a new download task that continues the existing download.

If the task cannot be resumed, your app should create a new download task and restart the transaction from the beginning.

In either case, if the transfer failed for any reason other than a server error, go to step 3 (creating and resuming task objects).

> **Note:** `NSURLSession` does not report server errors through the error parameter. The only errors your delegate receives through the error parameter are client-side errors, such as being unable to resolve the hostname or connect to the host. The error codes are described in `URL Loading System Error Codes`.
>
> Server-side errors are reported through the HTTP status code in the `NSHTTPURLResponse` object. For more information, read the documentation for the `NSHTTPURLResponse` and `NSURLResponse` classes.

14. If the response is multipart encoded, the session may call the delegate's `didReceiveResponse` method again, followed by zero or more additional `didReceiveData` calls. If this happens, go to step 7 (handling the `didReceiveResponse` call).
15. When you no longer need a session, invalidate it by calling either `invalidateAndCancel` (to cancel outstanding tasks) or `finishTasksAndInvalidate` (to allow outstanding tasks to finish before invalidating the object).

    After invalidating the session, when all outstanding tasks have been canceled or have finished, the session sends the delegate a `URLSession:didBecomeInvalidWithError:` message. When that delegate method returns, the session disposes of its strong reference to the delegate.

    > **Important:** The session object keeps a strong reference to the delegate until your app explicitly invalidates the session. If you do not invalidate the session, your app leaks memory.

If your app cancels an in-progress download, the `NSURLSession` object calls the delegate's `URLSession:task:didCompleteWithError:` method as though an error occurred.

Document Revision History

This table describes the changes to *URL Loading System Programming Guide*.

Date	Notes
2013-10-22	Made minor technical clarifications.
2013-09-18	Updated to describe the new NSURLSession API in OS X v10.9 and iOS 7.
2010-09-01	Fixed typos and removed deprecated symbols from code examples.
2010-03-24	Restructured content and added discussions of new authentication functionality.
2009-08-14	Added links to Cocoa Core Competencies.
2008-05-20	Updated to include content about NSURLDownload availability in iOS.
2008-05-06	Made minor editorial changes.
2007-07-10	Corrected minor typos.
2006-05-23	Added links to sample code.
2006-03-08	Updated sample code.
2005-09-08	Corrected connectionDidFinishLoading: method signature.
2005-04-08	Added accessor method to sample code. Corrected minor typos.
2004-08-31	Corrected minor typos. Corrected table of contents ordering.
2003-07-03	Corrected `willSendRequest:redirectResponse:` method signature throughout topic.

Date	Notes
2003-06-11	Added additional article outlining differences in behavior between NSURLDownload and NSURLConnection.
2003-06-06	First release of conceptual and task material covering the usage of new classes in Mac OS X v10.2 with Safari 1.0 for downloading content from the Internet.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010