Making HTTP and HTTPS Requests

OS X and iOS provide a number of general-purpose APIs for making HTTP and HTTPS requests. With these APIs, you can download files to disk, make simple HTTP and HTTPS requests, or precisely tune your request to the specific requirements of your server infrastructure.

When choosing an API, you should first consider why you are making an HTTP request:

- If you are writing a Newsstand app, you should use the `NKAssetDownload` API to download content in the background.
- If you need to download a file to disk in OS X, the easiest way is to use the `NSURLDownload` class. For details, see Downloading the Contents of a URL to Disk.
- You should use `CFHTTPStream` if any of the following are true:
    - You have a strict requirement not to use Objective-C.
    - You need to override proxy settings.
    - You need to be compatible with a particular non-compliant server.

  For more information, see Making Requests Using Core Foundation.
- Otherwise, you should generally use the `NSURLSession` or `NSURLConnection` APIs.

The sections below describe these APIs in more detail.

Note: If you have specific needs, you can also write your own HTTP client implementation using socket or socket-stream APIs. These APIs are described in Using Sockets and Socket Streams.

Making Requests Using Foundation

The following tasks describe common operations with the `NSURLSession` class, the `NSURLConnection` class, and related classes.

Retrieving the Contents of a URL without Delegates

If you just need to retrieve the contents of a URL and do something with the results at the end, in OS X v10.9 and later or iOS 7 and later, you should use the `NSURLSession` class. You can also use the `NSURLConnection` class for compatibility with earlier versions of OS X and iOS.

To do this, call one of the following methods: `dataTaskWithRequest:completionHandler:` (`NSURLSession`), `dataTaskWithURL:completionHandler:` (`NSURLSession`), or `sendAsynchronousRequest:queue:completionHandler:` (`NSURLConnection`). Your app must provide the following information:

- As appropriate, either an `NSURL` object or a filled-out `NSURLRequest` object that provides the URL, body data, and any other information that might be required for your particular request.
- A completion handler block that runs whenever the transfer finishes or fails.
- For `NSURLConnection`, an `NSOperation` queue on which your block should run.

If the transfer 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 the URL loading system is unable to retrieve the contents of the URL, an `NSError` object is passed as the third parameter.

Retrieving the Contents of a URL with Delegates

If your app needs more control over your request, such as controlling whether redirects are followed, performing custom authentication, or obtaining the data piecewise as it is received, you can use the `NSURLSession` class with a custom delegate. For compatibility with earlier versions of OS X and iOS, you can also use the `NSURLConnection` class with a custom delegate.

For the most part, the `NSURLSession` and `NSURLConnection` classes work similarly at a high level. However, there are a few significant differences:

- The `NSURLSession` API provides support for download tasks that behave much like the `NSURLDownload` class. This usage is described further in Downloading the Contents of a URL to Disk.
- When you create an `NSURLSession` object, you provide a reusable configuration object that encapsulates many common configuration options. With `NSURLConnection`, you must set those options on each connection independently.
- An `NSURLConnection` object handles a single request and any follow-on requests.

  An `NSURLSession` object manages multiple tasks, each of which represents a single URL request and any follow-on requests. You usually create a session when your app launches, then create tasks in much the same way that you would create `NSURLConnection` objects.
- With `NSURLConnection`, each connection object has a separate delegate. With `NSURLSession`, the delegate is shared across all tasks within a session. If you need to use a different delegate, you must create a new session.

When you initialize an `NSURLSession` or `NSURLConnection` object, the connection or session is automatically scheduled in the current run loop in the default run loop mode.

The delegate you provide receives notifications throughout the connection process, including intermittent calls to the `URLSession:dataTask:didReceiveData:` or `connection:didReceiveData:` method when a connection receives additional data from the server. It is the delegate's responsibility to keep track of the data it has already received, if necessary. As a rule:

- If the data can be processed a piece at a time, do so. For example, you might use a streaming XML parser.
- If the data is small, you might append it to an `NSMutableData` object.
- If the data is large, you should write it to a file and process it upon completion of the transfer.

When the `URLSession:task:didCompleteWithError:` or `connectionDidFinishLoading:` method is called, the delegate has received the entirety of the URL's contents.

Downloading the Contents of a URL to Disk

In OS X v10.9 and later or iOS 7 and later, if you need to download a URL and store the results as a file, but do not need to process the data in flight, the `NSURLSession` class lets you download the URL directly to a file on disk in a single step (as opposed to loading the URL into memory and then writing it out yourself). The `NSURLSession` class also allows you to pause and resume downloads, restart failed downloads, and continue downloading while the app is suspended, crashed, or otherwise not running.

In iOS, the `NSURLSession` class also launches your app in the background whenever a download finishes so that you can perform any app-specific processing on the file.

Note: In older versions of OS X, you can also download files to disk with the `NSURLDownload` class. The `NSURLDownload` class does not provide the ability to download files while the app is not running.

In older versions of iOS, you must use an `NSURLConnection` object to download the data to memory, then write the data to a file yourself.

To use the `NSURLSession` class for downloading, your code must do the following:

1. Create a session with a custom delegate and the configuration object of your choice:
   - If you want downloads to continue while your app is not running, you must provide a background session configuration object (with a unique identifier) when you create the session.
   - If you do not care about background downloading, you can create the session using any of the provided session configuration object types.
2. Create and resume one or more download tasks within the session.
3. Wait until your delegate receives calls from the task or session. In particular, you must implement the `URLSession:downloadTask:didFinishDownloadingToURL:` method to do something with a file when the download finishes and the `URLSession:task:didCompleteWithError:` call to handle any errors.

Note: The above steps are a greatly simplified view; depending on your needs, you may wish for your session delegate to handle a number of other delegate methods for custom authentication, redirect handling, and so on.

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 Retrieving the Contents of a URL with Delegates). 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. This process is described in Encoding URL Data.
- 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.)

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.

Configuring Authentication

Performing authentication with `NSURLSession` and `NSURLConnection` is relatively straightforward. The way you do this depends on the class you use and on the version of OS X or iOS that you are targeting.

For the `NSURLSession` class, your delegate should implement the `URLSession:task:didReceiveChallenge:completionHandler:` method. In this method, you perform whatever operations are needed to determine how to respond to the challenge, then call the provided completion handler with a constant that indicates how the URL Loading System should proceed and, optionally, a credential to use for authentication purposes.

For the `NSURLConnection` class:

- In OS X v10.7 and newer or iOS 5 and newer, your delegate should implement the `connection:willSendRequestForAuthenticationChallenge:` method. This method must call a method on the sender (the `NSURLConnection` object) to tell it how to proceed.
- In earlier versions, your delegate should implement both the `connection:canAuthenticateAgainstProtectionSpace:` and `connection:didReceiveAuthenticationChallenge:` methods.

  The `connection:didReceiveAuthenticationChallenge:` method is equivalent to the `connection:willSendRequestForAuthenticationChallenge:` method in later versions, and calls a method on the sender (the `NSURLConnection` object) to tell it how to proceed.

  The `connection:canAuthenticateAgainstProtectionSpace:` method should return `YES` if `[protectionSpace authenticationMethod]` is any of `NSURLAuthenticationMethodDefault`, `NSURLAuthenticationMethodHTTPBasic`, `NSURLAuthenticationMethodHTTPDigest`, `NSURLAuthenticationMethodHTMLForm`, `NSURLAuthenticationMethodNegotiate`, or `NSURLAuthenticationMethodNTLM`.

Possible Responses to an Authentication Challenge

Regardless of which class you use, your authentication handler method must examine the authentication challenge and tell the URL Loading System how to proceed:

- To provide a credential for authentication, pass `NSURLSessionAuthChallengeUseCredential` as the disposition (for `NSURLSession`) or call `useCredential:forAuthenticationChallenge:` (for `NSURLConnection`).

  For information about creating a credential object, read Creating a Credential Object.

- To continue the request without providing authentication, pass `NSURLSessionAuthChallengeUseCredential` as the disposition with a `nil` credential (for `NSURLSession`) or call `continueWithoutCredentialForAuthenticationChallenge:` (for `NSURLConnection`).
- To cancel the authentication challenge, pass `NSURLSessionAuthChallengeCancelAuthenticationChallenge` as the disposition (for `NSURLSession`) or call `cancelAuthenticationChallenge:` (for `NSURLConnection`). If you cancel the authentication challenge, the stream delegate's error method is called.
- To tell the operating system to handle the challenge as it ordinarily would, pass `NSURLSessionAuthChallengePerformDefaultHandling` as the disposition (for `NSURLSession`) or call `performDefaultHandlingForAuthenticationChallenge:` (for `NSURLConnection`). If you request default handling, then the operating system sends any appropriate credentials that exist in the credentials cache.

  Note: The `performDefaultHandlingForAuthenticationChallenge:` method was not supported prior to OS X v10.7 or iOS 5.

- To reject a particular type of authentication during the negotiation process, with the intent to accept a different method, pass `NSURLSessionAuthChallengeRejectProtectionSpace` as the disposition (for `NSURLSession`) or call `rejectProtectionSpaceAndContinueWithChallenge:` (for `NSURLConnection`).

  Note: The `rejectProtectionSpaceAndContinueWithChallenge:` method was not supported prior to OS X v10.7 or iOS 5.

Creating a Credential Object

Within your delegate's `connection:willSendRequestForAuthenticationChallenge:` or `connection:didReceiveAuthenticationChallenge:` method, you may need to provide an `NSURLCredential` object that provides the actual authentication information.

- For simple login/password authentication, call `credentialWithUser:password:persistence:`.
- For certificate-based authentication, call `credentialWithIdentity:certificates:persistence:` with a `SecIdentityRef` object (which is usually obtained from the user's keychain by calling `SecItemCopyMatching`).

Further Information

To learn more about the `NSURLSession` API, read *URL Session Programming Guide*. For related sample code, see *SimpleURLConnections*, *AdvancedURLConnections*, and *SeismicXML: Using NSXMLParser to parse XML documents*.

For details about the `NSURLConnection` API, read *URL Session Programming Guide*.

To learn more about using the `NSStream` API for making HTTP requests, read Setting Up Socket Streams in *Stream Programming Guide*.

For an example of the `setHTTPBodyStream:` method and the `CFStreamCreateBoundPair` function, see *SimpleURLConnections* in the iOS library. (The sample as a whole is designed to build and run on iOS, but the networking portions of the code are also useful on OS X.)

Making Requests Using Core Foundation

Other than the syntax details, the request functionality in Core Foundation is closely related to what is available at the Foundation layer. Thus, the examples in Making Requests Using Foundation should be helpful in understanding how to make requests using the `CFHTTPStream` API.

The Core Foundation URL Access Utilities are a C-language API that is part of the Core Foundation framework. To learn more, read *Core Foundation URL Access Utilities Reference*.

The `CFHTTPStream` API is a C-language API that is part of the Core Foundation framework. (You can, of course, use it in Objective-C code.) To learn more, read Communicating with HTTP Servers and Communicating with Authenticating HTTP Servers in *CFNetwork Programming Guide*.

These APIs are the most flexible way to communicate with an HTTP server (short of using sockets or socket streams directly), providing complete control over the message body as sent to the remote server, and control over most of the message headers as well. These APIs are also more complex, and thus should be used only if higher-level APIs cannot support your needs—for example, if you need to override the default system proxies.

Working with Web Services

If you are incorporating client-side web services communication in your OS X program, you can take advantage of a number of technologies:

- The `NSJSONSerialization` class converts between native Cocoa objects and JavaScript Object Notation (JSON).
- The `NSXMLParser` class provides a Cocoa API for SAX-style (streaming) parsing of XML content.
- The libxml2 library provides a cross-platform C API for SAX-style (streaming) and DOM-style (tree-based) parsing of XML content. For libxml2 documentation, see http://xmlsoft.org/.
- The `NSXMLDocument` API (in OS X only) provides DOM-style support for XML content.

In addition, a number of third-party libraries exist for working with web services.

Important: The Web Services Core framework is deprecated and should not be used for new development.