# 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:

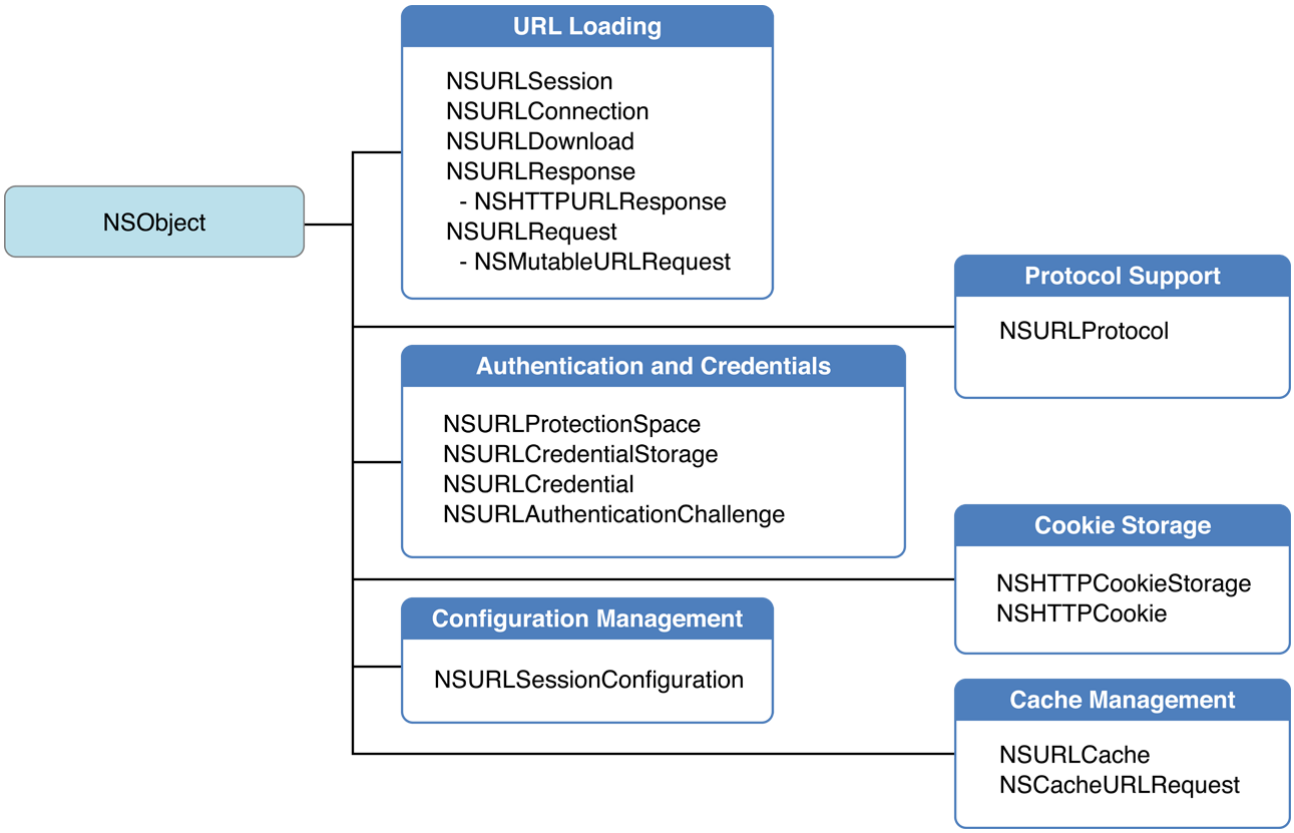
- File Transfer Protocol (`ftp://`)
- Hypertext Transfer Protocol (`http://`)
- Hypertext Transfer Protocol with encryption (`https://`)
- Local file URLs (`file:///`)
- Data URLs (`data://`)

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, Using NSURLConnection

### 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, Using NSURLDownload

## 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 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

## 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

## 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

## 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

## 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

# 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. 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. Then read Life Cycle of a URL Session 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.
- For using `NSURLDownload` to download files asynchronously to disk, read Using NSURLDownload.

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 explains how to encode arbitrary strings to make them safe for use in a URL.
- Handling Redirects and Other Request Changes describes the options you have for responding to a change to your URL request.
- Authentication Challenges and TLS Chain Validation describes the process for authenticating your connection against a secure server.
- Understanding Cache Access describes how a connection uses the cache during a request.
- Cookies and Custom Protocols 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.