

URL Loading System Programming Guide



Developer

Contents

Introduction 5

Organization of This Document 5

See Also 6

URL Loading System Overview 7

URL Loading 7

Cache Management 9

Authentication and Credentials 9

Cookie Storage 10

Protocol Support 11

Using NSURLConnection 12

Creating a Connection 12

Controlling Response Caching 15

Estimating Upload Progress 16

Downloading Data Synchronously 17

Using NSURLDownload 18

Downloading to a Predetermined Destination 18

Downloading a File Using the Suggested Filename 20

Displaying Download Progress 22

Resuming Downloads 24

Decoding Encoded Files 24

Handling Redirects and Other Request Changes 26

Authentication Challenges 28

Deciding How to Respond to an Authentication Challenge 28

Responding to an Authentication Challenge 29

 Providing Credentials 29

 Continuing Without Credentials 30

 Canceling the Connection 30

Understanding Cache Access 32

Using the Cache for a Request	32
Cache Use Semantics for the http Protocol	33
Document Revision History	34

Figures and Listings

URL Loading System Overview 7

Figure 1 The URL loading system class hierarchy 7

Using `NSURLConnection` 12

- Listing 1 Creating a connection using `NSURLConnection` 12
- Listing 2 Example connection:`didReceiveResponse:` implementation 13
- Listing 3 Example connection:`didReceiveData:` implementation 14
- Listing 4 Example connection:`didFailWithError:` implementation 14
- Listing 5 Example connection:`didFinishLoading:` implementation 15
- Listing 6 Example connection:`withCacheResponse:` implementation 16

Using `NSURLDownload` 18

- Listing 1 Using `NSURLDownload` with a predetermined destination file location 18
- Listing 2 Using `NSURLDownload` with a filename derived from the download 20
- Listing 3 Logging the finalized filename using `download:didCreateDestination:` 22
- Listing 4 Displaying the download progress 22
- Listing 5 Example implementation of `download:shouldDecodeSourceDataOfMIMEType: method.` 24

Handling Redirects and Other Request Changes 26

- Listing 1 Example of an implementation of `connection:willSendRequest:redirectResponse:` 26

Authentication Challenges 28

- Listing 1 An example of using the `connection:didReceiveAuthenticationChallenge:` `delegate` method 30

Introduction

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 `NSURL` class provides the ability to manipulate URLs and the resources they refer to.

The Foundation framework also provides a rich collection of classes that include support for URL loading, cookie storage, response caching, credential storage and authentication, and writing custom protocol extensions.

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

- File Transfer Protocol (`ftp://`)
- Hypertext Transfer Protocol (`http://`)
- Secure 128-bit Hypertext Transfer Protocol (`https://`)
- Local file URLs (`file:///`)

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

Organization of This Document

This guide includes the following articles:

- [“URL Loading System Overview”](#) (page 7) describes the classes of the URL loading system and their interaction.
- [“Using `NSURLConnection`”](#) (page 12) describes using `NSURLConnection` for asynchronous connections.
- [“Using `NSURLDownload`”](#) (page 18) describes using `NSURLDownload` to download files asynchronously to disk.
- [“Handling Redirects and Other Request Changes”](#) (page 26) describes the options you have for responding to a change to your URL request.
- [“Authentication Challenges”](#) (page 28) describes the process for authenticating your connection against a secure server.
- [“Understanding Cache Access”](#) (page 32) describes how a connection uses the cache during a request.

See Also

The following sample code is available through Apple Developer Connection:

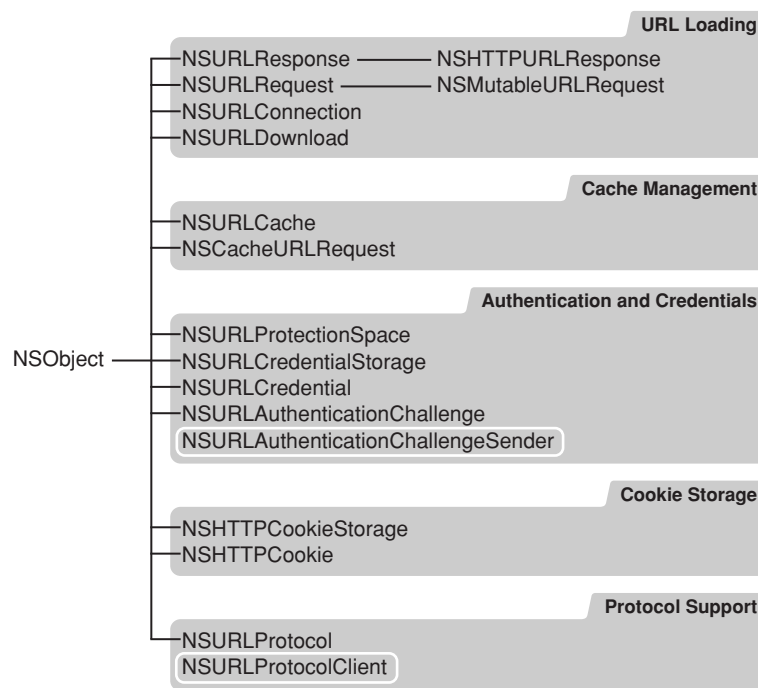
- *SpecialPictureProtocol* implements a custom `NSURLProtocol` that creates jpeg images in memory as data is downloaded.
- *AutoUpdater* demonstrates how to check for, and download, an application update using `NSURLConnection` and `NSURLDownload`.

URL Loading System Overview

The URL loading system is a set of classes and protocols that provide the underlying capability for an application to access the data specified by a URL.

These classes fall into five categories: URL loading, cache management, authentication and credentials, cookie storage, and protocol support.

Figure 1 The URL loading system class hierarchy



URL Loading

The most commonly used classes in the URL loading system allow an application to create a request for the content of a URL and download it from the source.

A request for the contents of a URL is represented by an `NSURLRequest` object. The `NSURLRequest` class encapsulates a URL and any protocol-specific properties, in a protocol-independent manner. It also provides an interface to set the timeout for a connection and specifies the policy regarding the use of any locally cached data. The `NSMutableURLRequest` class is a mutable subclass of `NSURLRequest` that allows a client application to alter an existing request.

Note: When a client application 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 once a download has been initialized.

Protocols, such as HTTP, that support protocol-specific properties must create categories on the `NSURLRequest` and `NSMutableURLRequest` classes to provide accessors for those properties. As an example, the HTTP protocol adds methods to `NSURLRequest` to return the HTTP request body, headers, and transfer method. It also adds methods to `NSMutableURLRequest` to set the corresponding values. Methods for setting and getting property values in those accessors are exposed in the `NSURLProtocol` class.

The response from a server to a request can be viewed as two parts: metadata describing the contents and the URL content data. The 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. Protocols can create subclasses of `NSURLResponse` to store protocol-specific metadata. `NSHTTPURLResponse`, for example, stores the headers and the status code returned by the web server.

Note: It's important to remember that only the metadata for the response is stored in an `NSURLResponse` object. An `NSCachedURLResponse` instance is used to encapsulate an `NSURLResponse`, the URL content data, and any application-provided information. See [“Cache Management”](#) (page 9) for details.

The `NSURLConnection` and `NSURLDownload` classes provide the interface to make a connection specified by an `NSURLRequest` object and download the contents. An `NSURLConnection` object provides data to the delegate as it is received from the originating source, whereas an `NSURLDownload` object writes the request data directly to disk. Both classes provide extensive delegate support for responding to redirects, authentication challenges, and error conditions.

The `NSURLConnection` class provides a delegate method that allows an application to control the caching of a response on a per-request basis. Downloads initiated by an `NSURLDownload` instance are not cached.

Cache Management

The URL loading system provides a composite on-disk and in-memory cache allowing an application to reduce its dependence on a network connection and provide faster turnaround for previously cached responses. The cache is stored on a per-application basis.

The cache is queried by `NSURLConnection` according to the cache policy specified by the initiating `NSURLRequest`.

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` and the URL content data. `NSCachedURLResponse` also provides a user info dictionary that can be used by an application to cache any custom data.

Not all protocol implementations support response caching. Currently only `http` and `https` requests are cached, and `https` requests are never cached to disk.

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

Authentication and Credentials

Some servers restrict access to certain content, requiring a user to authenticate with a valid user name and password in order to gain access. In the case of a web server, restricted content is grouped together into a realm that requires a single set of credentials.

The URL loading system provides classes that model credentials and protected areas as well as providing secure credential persistence. Credentials can be specified to persist for a single request, for the duration of an application'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 applications.

The `NSURLCredential` class encapsulates a credential consisting of the user name, password, and the type of persistence to use. 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.

Cookie Storage

Due to the stateless nature of the HTTP protocol, cookies are often used to provide persistent storage of data across URL requests. The URL loading system provides interfaces to create and manage cookies as well as sending and receiving cookies from web servers.

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

iOS Note: Cookies are not shared by applications in iOS.

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

Note: Changing the cookie acceptance policy in an application affects the cookie acceptance policy for all other running applications.

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

Protocol Support

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

Custom protocols are implemented by subclassing `NSURLProtocol` and then registering the new class with the URL loading system using the `NSURLProtocol` class method `registerClass:`. When an `NSURLConnection` or `NSURLDownload` object initiates a connection for an `NSURLRequest`, 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.

The URL loading system is responsible for creating and releasing `NSURLProtocol` instances when connections start and complete. An application 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.

Using `NSURLConnection`

`NSURLConnection` provides the most flexible method of downloading the contents of a URL. It 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

In order to download the contents of a URL, an application needs to provide a delegate object that, at a minimum, implements the following delegate methods: `connection:didReceiveResponse:`, `connection:didReceiveData:`, `connection:didFailWithError:` and `connectionDidFinishLoading:`.

The example in Listing 1 initiates a connection for a URL. It 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`. If the connection is successful, an instance of `NSMutableData` is created to store the data that is provided to the delegate incrementally.

Listing 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 connection with the request
// and start loading the data
NSURLConnection *theConnection=[[NSURLConnection alloc] initWithRequest:theRequest
delegate:self];
if (theConnection) {
    // Create the NSMutableData to hold the received data.
    // receivedData is an instance variable declared elsewhere.
```

```
        receivedData = [[NSMutableData data] retain];  
    } else {  
        // Inform the user that the connection failed.  
    }  
}
```

The download 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` 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 message can be sent due to server redirects, or in rare cases multi-part MIME documents. Each time the delegate receives the `connection:didReceiveResponse:` message, it should reset any progress indication and discard all previously received data. The example implementation in Listing 2 simply resets the length of the received data to 0 each time it is called.

Listing 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.  
  
    // 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 3, the new data is appended to the `NSMutableData` object created in Listing 1.

Listing 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.

If an error is encountered during the download, 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 message `connection:didFailWithError:`, it receives no further delegate messages for the specified connection.

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

Listing 4 Example connectionDidFailWithError: implementation

```
- (void)connection:(NSURLConnection *)connection
didFailWithError:(NSError *)error
{
    // release the connection, and the data object
    [connection release];
    // receivedData is declared as a method instance elsewhere
    [receivedData release];

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

Finally, if the connection succeeds in downloading the request, the delegate receives the `connectionDidFinishLoading:` message. The delegate will receive no further messages for the connection and the `NSURLConnection` object can be released.

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

Listing 5 Example `connectionDidFinishLoading:` implementation

```
- (void)connectionDidFinishLoading:(NSURLConnection *)connection
{
    // do something with the data
    // responseData is declared as a method instance elsewhere
    NSLog(@"Succeeded! Received %d bytes of data",[responseData length]);

    // release the connection, and the data object
    [connection release];
    [responseData release];
}
```

This 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 caching of the response.

Controlling Response Caching

By default the data for a connection is cached according to the support provided by the `NSURLProtocol` subclass that handles the request. An `NSURLConnection` delegate can further refine that behavior by implementing `connection:willCacheResponse:`.

This delegate method can examine the provided `NSCachedURLResponse` object and change how the response is cached, for example restricting its storage to memory only or preventing it from being cached altogether. It is also possible to insert objects in an `NSCachedURLResponse`'s user info dictionary, causing them to be stored in the cache as part of the response.

Note: The delegate receives `connection:willCacheResponse:` messages only for protocols that support caching.

The example in Listing 6 prevents the caching of `https` responses. It also adds the current date to the user info dictionary for responses that are cached.

Listing 6 Example `connection:withCacheResponse:` implementation

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

    if ([[[[cachedResponse response] URL] scheme] isEqual:@"https"]) {
        newCachedResponse = nil;
    } else {
        NSDictionary *newUserInfo;
        newUserInfo = [NSDictionary dictionaryWithObject:[NSDate date]
                                                    forKey:@"Cached Date"];

        newCachedResponse = [[NSCachedURLResponse alloc]
                               initWithResponse:[cachedResponse response]
                               data:[cachedResponse data]
                               userInfo:newUserInfo
                               storagePolicy:[cachedResponse storagePolicy]]
                               autorelease];
    }
    return newCachedResponse;
}
```

Estimating Upload Progress

You can estimate the progress of an HTTP POST upload with the `connection:didSendBodyData:totalBytesWritten:totalBytesExpectedToWrite:` delegate method. Note that this is not an exact measurement of upload progress, because the connection may fail or the connection may encounter an authentication challenge.

Downloading Data Synchronously

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

- The client application blocks until the data has been completely received, an error is encountered, or the request times out.
- 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.

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

If the request requires authentication in order to make the connection, valid credentials must already be available in the `NSURLCredentialStorage`, 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

NSURLDownload provides 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 NSURLConnection class instead. See [“Using NSURLConnection”](#) (page 12) 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 1 sets the destination, and thus requires the delegate only implement the `download:didFailWithError:` and `downloadDidFinish:` methods.

Listing 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
{
    // Release the connection.
    [download release];

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

- (void)downloadDidFinish:(NSURLDownload *)download
{
    // Release the connection.
    [download release];

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

Additional methods can be implemented by the delegate 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 2 saves the downloaded file to the desktop using the suggested filename.

Listing 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
{
    // Release the download.
    [download release];

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

- (void)downloadDidFinish:(NSURLDownload *)download
{
    // Release the download.
    [download release];

    // 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 logs the finalized filename.

Listing 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

The progress of the download can be determined 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 this message is received.

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

Listing 4 Displaying the download progress

```
- (void)setDownloadResponse:(NSURLResponse *)aDownloadResponse
{
    [aDownloadResponse retain];

    // downloadResponse is an instance variable defined elsewhere.
    [downloadResponse release];
    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;

    // Retain 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 `N0` 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 selected file formats: MacBinary, BinHex and gzip. 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 5 compares the MIME type of the file and allows decoding of MacBinary and BinHex encoded content.

Listing 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/gzip"]) {
        shouldDecode = NO;
    }
    return shouldDecode;
}
```



```
}
```

Handling Redirects and Other Request Changes

A server may redirect a request for one URL to another URL. The delegates for `NSURLConnection` and `NSURLDownload` can be notified when this occurs for their connection.

To handle a redirect for an instance of `NSURLConnection`, implement the `connection:willSendRequest:redirectResponse:` delegate method (for `NSURLDownload`, implement `download:willSendRequest:redirectResponse:`). If the delegate implements this method, it can examine the new request and the response that caused the redirect, and respond in one of four ways:

- The delegate can allow the redirect by simply returning the provided request.
- The delegate can create a new request, pointing to a different URL, and return that request.
- The delegate can reject the redirect and receive any existing data from the connection by returning `nil`.
- The delegate can cancel both the redirect and the connection by sending the `cancel` message to the `NSURLConnection` or `NSURLDownload`.

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 1 allows canonical changes and denies all server redirects.

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

```
-(NSURLRequest *)connection:(NSURLConnection *)connection
    willSendRequest:(NSURLRequest *)request
    redirectResponse:(NSURLResponse *)redirectResponse
{
    NSURLRequest *newRequest = request;
    if (redirectResponse) {
        newRequest = nil;
    }
}
```

```
        return newRequest;  
    }
```

If the delegate doesn't implement `connection:willSendRequest:redirectResponse:`, all canonical changes and server redirects are allowed.

Authentication Challenges

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

Deciding How to Respond to an Authentication Challenge

If an `NSURLRequest` object requires authentication, the delegate of the `NSURLConnection` (or `NSURLDownload`) object associated with the request first 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.

Note: If your delegate does not implement the `connection:canAuthenticateAgainstProtectionSpace:` method and the protection space uses client certificate authentication or server trust authentication, the system behaves as if you returned `NO`. The system behaves as if you returned `YES` for all other authentication methods.

If your delegate returns `YES` from `connection:canAuthenticateAgainstProtectionSpace:` or doesn't implement it, and there are no valid credentials available, either as part of the requested URL or in the shared `NSURLCredentialStorage`, the delegate receives a `connection:didReceiveAuthenticationChallenge:` message. 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, 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 end 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`)

The basic authentication method 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, digest authentication just 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`)

Client certificate authentication 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)

Server trust authentication 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, pass it 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 by calling `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 by calling `cancelAuthenticationChallenge: on [challenge sender]`. The delegate receives a `connection:didCancelAuthenticationChallenge:` message, providing the opportunity to give the user feedback.

Para

The implementation shown in Listing 1 attempts to authenticate 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 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.

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 will cause the URL loading system to use cached data ignoring its age or expiration date, if it exists, and 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 provide caching if they choose.

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 data is fetched from the originating source. If there is a cached response 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 a connection is made to the originating source to see if it has changed. If it has not changed, then the response is returned from the local cache. If it has changed, the data is fetched from the originating source.

If the cached response doesn't specify that the contents must be revalidated, the maximum age or expiration specified in the response is examined. If the cached response is recent enough, then the response is returned from the local cache. If the response is determined to be stale, the originating source is checked for newer data. If newer data is available, the data is fetched from the originating source, otherwise it is returned from the cache.

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

Document Revision History

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

Date	Notes
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 <code>connectionDidFinishLoading:</code> 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 <code>willSendRequest:redirectResponse:</code> method signature throughout topic.
2003-06-11	Added additional article outlining differences in behavior between NSURLDownload and NSURLConnection.

Date	Notes
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.
© 2003, 2010 Apple Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

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

Apple, the Apple logo, Cocoa, Mac, Mac OS, OS X, and Safari are trademarks of Apple Inc., registered in the U.S. and other countries.

iOS is a trademark or registered trademark of Cisco in the U.S. and other countries and is used under license.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.