URL Loading System

Cocoa > Networking

2008-05-06

Contents

Figures and Listings

Introduction to the URL Loading System

This programming topic describes the Foundation framework classes available for interacting with URLs and communicating with servers using standard Internet protocols. Together theses 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 programming topic includes the following articles:

- "URL Loading System Overview" (page 9) describes the classes of the URL loading system and their interaction.
- "Downloading Data Synchronously" (page 13) describes the NSURLConnection support for synchronous connections.
- "Using NSURLConnection" (page 15) describes using NSURLConnection for asynchronous connections.
- "Using NSURLDownload" (page 21) describes using NSURLDownload to download files asynchronously to disk.
- "Understanding Cache Access" (page 29) describes how a connection uses the cache during a request.
- "NSURLDownload and NSURLConnection Differences" (page 31) describes subtle differences in behavior between these two classes.

See Also

The following sample code is available through Apple Developer Connection:

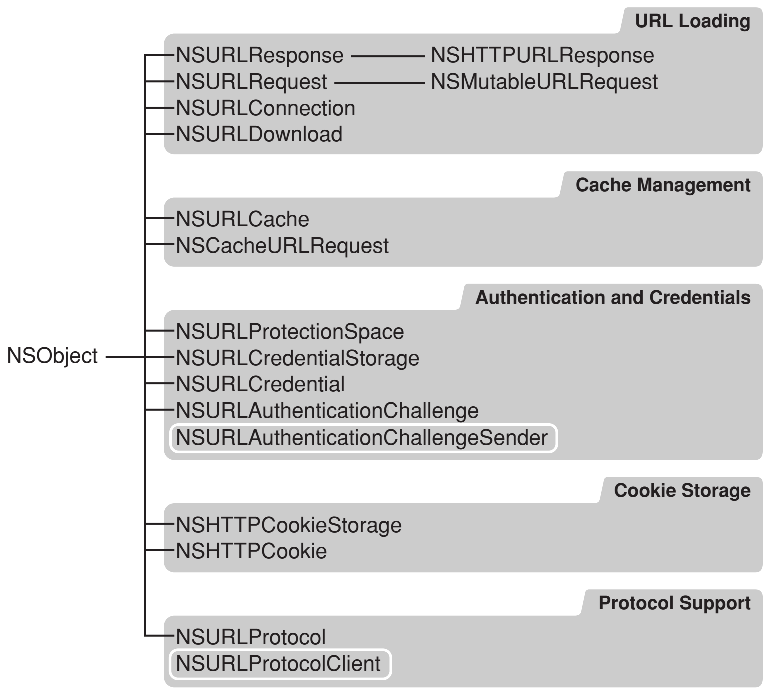
- *SpecialPictureProtocol* implements a custom `NSURLProtocol` that creates jpeg images in memory as data is downloaded.
- *AutoUpdater* demonstrates how to check for, and download, an application update using `NSURLConnection` and `NSURLDownload`.

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 10) 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 or NSURLDownload 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 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.

Note: Cookies are shared among all applications using the URL loading system.

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.

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.

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 simple and convenient, but has 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 is 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 will return `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 will respond 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 NSURLConnection

NSURLConnection provides the most flexible method of downloading the contents of a URL. It provides a simple interface for creating and cancelling 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 timeout interval for the connection. It then creates an NSURLConnection instance using the request and specifying the 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 will be 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 that will hold
    // the received data
    // receivedData is declared as a method instance elsewhere
    receivedData=[[NSMutableData data] retain];
} else {
    // inform the user that the download could not be made
}
```

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.

It's important that the delegate be prepared to receive the `connection:didReceiveResponse:` message multiple times for a 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 declared as a method instance elsewhere
    [receivedData setLength:0];
}
```

As the connection progresses the delegate is 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 the receivedData
    // receivedData is declared as a method instance 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 `NSErrorFailingURLStringKey`.

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:NSErrorFailingURLStringKey]);
}
```

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

    // release the connection, and the data object
    [connection release];
    [receivedData 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.

Handling Request Changes

It's not uncommon for a server to redirect a request for one URL to another URL. The NSURLConnection delegate will receive a `connection:willSendRequest:redirectResponse:` when this occurs.

If the delegate implements this method it can examine the new NSURLRequest and the NSURLResponse that caused the redirect and allow the redirected NSURLRequest to be used for the connection, create a new NSURLRequest for the connection, reject the redirect and have the connection return any data received from the NSURLResponse that caused the redirect, or cancel the download entirely.

To allow the redirect, the delegate should return the provided NSURLRequest. The delegate could also create a new NSURLRequest, pointing to a new URL, and return that request.

If the delegate wishes to reject the redirect, but receive any existing data for the connection, the method should return `nil`.

Finally, the delegate can cancel the redirect and the connection, by sending the `cancel` message to `connection`.

The delegate will also receive this 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 is required 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 NSURLRequest.

The example implementation in Listing 6 allows canonical changes and denies all server redirects.

Listing 6 Example connection:willSendRequest:redirectResponse: implementation.

```
-(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.

Handling Authentication Challenges

If a request requires authentication and there are no valid credentials available, either as part of the requested URL or in the shared NSURLCredentialStorage, the NSURLConnection delegate receives a `connection:didReceiveAuthenticationChallenge:` message. In order for the connection to continue, the delegate must provide credentials to attempt to use for authentication, attempt to continue without credentials, or cancel the authentication request.

The NSURLAuthenticationChallenge instance passed to the delegate contains information about what triggered the authentication challenge, the number of attempts that have been made for the challenge, any attempted credentials, the NSURLProtocolSpace that requires the credentials, and the sender of the challenge.

Often the delegate prompts the user to enter a valid user name and password. If the authentication challenge has tried to authenticate and failed, the attempted credentials are returned by sending `challenge` a `proposedCredential` message. The delegate can then use these credentials to populate a dialog that it presents to the user.

Invoking `previousFailureCount` on the challenge parameter returns the number of authentication attempts. The delegate can provide this information to the end user, to determine if the credentials it supplied previously are failing, or to limit the maximum number of authentication attempts.

To attempt to authenticate, the application should create an NSURLCredential object with the user name, password and the type of persistence to use for the credentials, and then send the `[challenge sender]` a `useCredential:forAuthenticationChallenge:` message.

If the delegate chooses not to provide a credential for the authentication challenge, it can attempt to continue without one by sending `[challenge sender]` a `continueWithoutCredentialsForAuthenticationChallenge:` message. Depending on the protocol implementation, this may return alternate URL contents that don't require authentication or cause the connection to fail, receiving a `connectionDidFailWithError:` message.

The delegate may also choose to cancel the authentication challenge by sending `cancelAuthenticationChallenge:` to `[challenge sender]`. The delegate receives a `connection:didCancelAuthenticationChallenge:` message providing the opportunity to give the user feedback.

The example implementation in Listing 7 attempts to authenticate the challenge by creating an NSURLCredential instance using 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 7 Example of 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 NSURLCredentialStorage or must be provided as part of the requested URL. If the credentials are not available or fail to authenticate, a `continueWithoutCredentialForAuthenticationChallenge:` message is sent by the underlying implementation.

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, perhaps 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 8 prevents the caching of `https` responses. It also adds the current date to the user info dictionary for responses that are cached.

Listing 8 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:[NSCalendarDate date]
                                                forKey:@"Cached Date"];
        newCachedResponse=[[[NSCachedURLResponse alloc]
                            initWithResponse:[cachedResponse response]
                    data:[cachedResponse data]
                        userInfo:newUserInfo
                               storagePolicy:[cachedResponse storagePolicy]]
                           autorelease];
    }
    return newCachedResponse;
}
```

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.

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

Downloading to a Predetermined Destination

One usage pattern for NSURLDownload is downloading a file to a predetermined filename on the disk. If the application knows the destination of the download, it can explicitly set it 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:CONFIG_SOURCE_URL_STRING]

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 now
        [theDownload setDestination:CONFIG_SOURCE_PATH allowOverwrite:YES];
    } else {
        // inform the user that the download could not be made
    }
}
```

```
- (void)download:(NSURLDownload *)download didFailWithError:(NSError *)error
{
    // release the connection
    [download release];

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

- (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

Another common situation is that 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 a users 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 connection with the request
    // and start loading the data
NSURLDownload  *theDownload=[[NSURLDownload alloc] initWithRequest:theRequest
delegate:self];
    if (!theDownload) {
        // inform the user that the download could not be made
    }
}

- (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 connection
    [download release];

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

- (void)downloadDidFinish:(NSURLDownload *)download
{
    // release the connection
    [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 the 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 release];
    downloadResponse = aDownloadResponse;
}

- (void)download:(NSURLDownload *)download didReceiveResponse:(NSURLResponse
*)response
{
    // reset the progress, this might be called multiple times
    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.

Handling Request Changes

It’s not uncommon for a server to redirect a request for one URL to another URL. The NSURLDownload delegate receives a `download:willSendRequest:redirectResponse:` when this occurs.

If the delegate implements this method, it can examine the new NSURLRequest and the NSURLResponse that caused the redirect and allow the redirected NSURLRequest to be used for the download, create a new NSURLRequest for the download, reject the redirect and return any data received from the NSURLResponse that caused the redirect, or cancel the download entirely.

To allow the redirect to occur, the delegate implementation should return the NSURLRequest passed to the delegate method. The delegate could also create a new NSURLRequest, pointing to a new URL, and return that request.

If the delegate wishes to reject the redirect, but receive any existing data for the connection, it should return `nil`.

Finally, the delegate can cancel the redirect and the connection by calling `[connection cancel]`.

The delegate also receives this 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 is required 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 NSURLRequest.

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

Listing 5 Example download:willSendRequest:redirectResponse: implementation.

```
-(NSURLRequest *)download:(NSURLDownload *)download
            willSendRequest:(NSURLRequest *)request
           redirectResponse:(NSURLResponse *)redirectResponse
{
    NSURLRequest *newRequest=request;
    if (redirectResponse) {
        newRequest=nil;
    }
    return newRequest;
}
```

If the delegate doesn’t implement `download:willSendRequest:redirectResponse:`, the default behavior is to allow all canonical changes and server redirects.

Handling Authentication Challenges

If a request requires authentication and there are no valid credentials available, either as part of the requested URL or in the shared NSURLCredentialStorage, the NSURLDownload delegate receives a `download:didReceiveAuthenticationChallenge:` message. In order for the download to continue, the delegate must provide credentials, attempt to continue without credentials, or cancel the authentication request.

The NSURLAuthenticationChallenge (the challenge) passed to the delegate contains information about what triggered the authentication challenge, the number of attempts that have been made for the challenge, any attempted credentials, the NSURLProtocolSpace that requires the credentials, and the sender of the challenge.

If the authentication challenge has tried to authenticate and failed, the attempted credentials are returned by calling `[challenge proposedCredential]`. The delegate can then use the previously attempted credential to populate a dialog and prompt the user.

The number of attempts at authentication for the challenge is returned by calling `[challenge previousFailureCount]`. The delegate can pass this information along to the end user, to determine if the credentials supplied previously are failing, or to limit the maximum number of authentication attempts.

To attempt to authenticate, the application should create an NSURLCredential object with the user name, password and the type of persistence to use for the credentials, and then send the `[challenge sender]` a `useCredential:forAuthenticationChallenge:` message.

If the delegate chooses not to provide a credential for the authentication challenge it can attempt to continue without one by sending `[challenge sender]` a `continueWithoutCredentialsForAuthenticationChallenge:` message. Depending on the protocol implementation, this may return alternate URL contents that don't require authentication or cause the connection to fail, receiving a `connectionDidFailWithError:` message.

The delegate may choose to cancel the authentication challenge by sending `cancelAuthenticationChallenge:` to `[challenge sender]`. The delegate then receives a `connection:didCancelAuthenticationChallenge:` message, providing the opportunity to give the end user feedback.

The example in Listing 6 attempts to authenticate the challenge by creating an NSURLCredential instance using 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 Example download:didReceiveAuthenticationChallenge: implementation.

```
-(void)download:(NSURLDownload *)download
        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];
    }
}
```

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 7 compares the MIME type of the file and allows decoding of MacBinary and BinHex encoded content.

Listing 7 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;
}
```

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.

NSURLDownload and NSURLConnection Differences

Aside from the obvious capability differences, NSURLDownload and NSURLConnection have two more subtle, yet significant, differences in behavior.

Response Caching

NSURLConnection provides support for caching of a response from a server in the application's NSURLCache storage. It also provides the delegate method `connection:shouldCacheResponse:` which allows an application to customize the cached response.

NSURLDownload does not cache responses in the application's NSURLCache storage, nor does it provide a delegate method to enable this behavior.

Handling of Non-existent URLs

One of the significant differences between NSURLConnection and NSURLDownload is the handling of requests for non-existent URLs when web server has be configured to return an alternate page in response to an error.

When using NSURLConnection the default behavior is to allow the redirect and return the contents of the redirected URL, instead of returning an error. If an attempt is made to download the same URL using NSURLDownload, an error is returned indicating that the file has not been found.

An NSURLConnection delegate can mimic the NSURLDownload behavior by implementing the `connection:willSendRequest:responseRequest:` method, and examining the provided NSURLResponse. If the response is an NSHTTPURLResponse object, and the *`statusCode`* is 4xx or 5xx, then the server is attempting to redirect to an alternate error page. The delegate can prevent this by returning `nil`.

Document Revision History

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

Date	Notes
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.
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.