Using NSURLConnection

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

Creating a Connection

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

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

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

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

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

Listing 2-1 Creating a connection using `NSURLConnection`

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


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


// create the connection with the request
// and start loading the data
NSURLConnection *theConnection=[[NSURLConnection alloc] initWithRequest:theRequest
delegate:self];
```

```
if (!theConnection) {
    // Release the receivedData object.
    receivedData = nil;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Listing 2-5 Example `connectionDidFinishLoading:` implementation

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

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

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

Making a POST Request

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

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

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

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

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

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

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

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


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


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


// Designate the request a POST request and specify its body data
[postRequest setHTTPMethod:@"POST"];
```

```
[postRequest setHTTPBody:[NSData dataWithBytes:[bodyData UTF8String]
length:strlen([bodyData UTF8String])]];


// Initialize the NSURLConnection and proceed as described in

// Retrieving the Contents of a URL
```

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

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

Retrieving Data Using a Completion Handler Block

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

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

Note: This method has two significant limitations:

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

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

Retrieving Data Synchronously

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

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

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

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

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

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

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