Creating and running requests

Creating a synchronous request

The simplest way to use ASIHTTPRequest. Sending the **startSynchronous** message will execute the request in the same thread, and return control when it has completed (successfully or otherwise).

Check for problems by inspecting the **error** property.

To get the response as a string, call the **responseString** method. Don't use this for binary data - use **responseData** to get an NSData object, or, for larger files, set your request to download to a file with the **downloadDestinationPath** property.

```
- (IBAction)grabURL:(id)sender
{
  NSURL *url = [NSURL URLWithString:@"http://allseeing-i.com"];
  ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
  [request startSynchronous];
  NSError *error = [request error];
  if (!error) {
    NSString *response = [request responseString];
  }
}
```

In general, you should use asynchronous requests in preference to synchronous requests. When you use ASIHTTPRequest synchronously from the main thread, your application's user interface will lock up and become unusable for the duration of the request.

Creating an asynchronous request

Does the same thing as the previous example, but the request runs in the background.

```
- (IBAction)grabURLInBackground:(id)sender
{
   NSURL *url = [NSURL URLWithString:@"http://allseeing-i.com"];
   ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
   [request setDelegate:self];
   [request startAsynchronous];
}

- (void)requestFinished:(ASIHTTPRequest *)request
{
   // Use when fetching text data
   NSString *responseString = [request responseString];

   // Use when fetching binary data
   NSData *responseData = [request responseData];
}

- (void)requestFailed:(ASIHTTPRequest *)request
{
   NSError *error = [request error];
}
```

Note that we set the delegate property of the request so we can receive notification when the request completes or fails.

This is the simplest way to create an asynchronous request, and it will run in a global NSOperationQueue behind the scenes. For more complex operations (such as tracking progress across multiple requests), you might want to create your own queue, which is what we'll cover next.

Using blocks

As of v1.8, we can do the same thing using blocks, on platforms that support them:

```
- (IBAction)grabURLInBackground:(id)sender
{
   NSURL *url = [NSURL URLWithString:@"http://allseeing-i.com"];
   __block ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
   [request setCompletionBlock:^{
      // Use when fetching text data
      NSString *responseString = [request responseString];

      // Use when fetching binary data
      NSData *responseData = [request responseData];
   }];
   [request setFailedBlock:^{
      NSError *error = [request error];
   }];
   [request startAsynchronous];
}
```

Note the use of the *__block* qualifier when we declare the request, this is important! It tells the block not to retain the request, which is important in preventing a retain-cycle, since the request will always retain the block.

Using a queue

This example does the same thing again, but we've created an NSOperationQueue for our request.

Using an NSOperationQueue (or ASINetworkQueue, see below) you create yourself gives you more control over asynchronous requests. When using a queue, only a certain number of requests can run at the same time. If you add more requests than the queue's **maxConcurrentOperationCount** property, requests will wait for others to finish before they start.

```
- (IBAction)grabURLInTheBackground:(id)sender
{
   if (![self queue]) {
      [self setQueue:[[[NSOperationQueue alloc] init] autorelease]];
   }

   NSURL *url = [NSURL URLWithString:@"http://allseeing-i.com"];
   ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
   [request setDelegate:self];
   [request setDidFinishSelector:@selector(requestDone:)];
   [request setDidFailSelector:@selector(requestWentWrong:)];
   [[self queue] addOperation:request]; //queue is an NSOperationQueue
}

- (void)requestDone:(ASIHTTPRequest *)request
{
   NSString *response = [request responseString];
}

- (void)requestWentWrong:(ASIHTTPRequest *)request
{
   NSError *error = [request error];
}
```

In the above sample, 'queue' is a retained NSOperationQueue property of our controller.

We're setting custom selectors that will be called when the request succeeds or fails. If you don't set these, the defaults (**requestFinished:** and **requestFailed:**) will be used, as in the previous example.

Handling success and failure for multiple requests in delegate methods

If you need to handle success and failure on many different types of request, you have several options:

1. If your requests are all of the same broad type, but you want to distinguish between them, you can set the **userInfo** NSDictionary property of each request with your own custom data that you can read in your finished / failed delegate methods. For simpler cases, you can set the request's **tag** property instead. Both of these properties are for your own use, and are not sent to the server.
2. If you need to handle success and failure in a completely different way for each request, set a different setDidFinishSelector / setDidFailSelector for each request
3. For more complex situations, or where you want to parse the response in the background, create a minimal subclass of ASIHTTPRequest for each type of request, and override **requestFinished:** and **failWithError:**.

It's best to avoid using the request's URL to distinguish between different requests in your delegate methods, because the URL property can change when a request is redirected. If you really want to use the request URL, use [request originalURL] instead - this will always be the first url the request connected to.

About ASINetworkQueues

ASINetworkQueue is a subclass of NSOperationQueue that provides some additional functionality.

Its primary purpose is to allow you to track the upload and/or download progress of a whole queue (**Read more about progress tracking**).

In addition, ASINetworkQueues provide some extra delegate method selectors:

- **requestDidStartSelector**
  Called each time a request in the queue begins running. You can use this as an alternative to specifying a didStartSelector and setting a delegate on requests you add to the queue
- **requestDidReceiveResponseHeadersSelector**
  Called each time a request in the queue receives response headers from the server. For large downloads, this may be some time before the request actually completes. You can use this as an alternative to specifying a didReceiveResponseHeadersSelector and setting a delegate on requests you add to the queue
- **requestDidFinishSelector**
  Called each time a request in the queue completes successfully. You can use this as an alternative to specifying a didFinishSelector and setting a delegate on requests you add to the queue
- **requestDidFailSelector**
  Called each time a request in the queue fails. You can use this as an alternative to specifying a didFailSelector and setting a delegate on requests you add to the queue
- **queueDidFinishSelector**
  Called when the whole queue has completed, whether individual requests failed or succeeded.

To use these, set the delegate of the queue (rather than the delegate of the request) to the controller that implements the methods that these selectors represent.

ASINetworkQueues work slightly differently from NSOperationQueues, in that requests added to them will not start running immediately. When using an ASINetworkQueue, add all the operations you want to run, then call **[queue go]**. When you start a queue with **accurate progress turned on**, it will first perform a HEAD request for all the GET requests in the queue to get the total size of the data to be downloaded. Once it has the total size, it can accurately show the total progress, and the real requests will start.

What happens when you add a request to an ASINetworkQueue that has already been started?

If you are using an ASINetworkQueue to track the overall progress of several requests, overall progress will move backwards to take account of each new request *only when that request starts running*. An ASINetworkQueue will not perform a HEAD request for requests added after the queue is started, so if you are adding many requests to a running queue at once, the overall progress will not update immediately.

If a queue is already running, you do not need to call [queue go] again.

When a request in an ASINetworkQueue fails, the queue will by default cancel all other requests. You can disable this behaviour with **[queue setShouldCancelAllRequestsOnFailure:NO]**.

ASINetworkQueues can only run ASIHTTPRequest operations, they cannot be used for generic operations. Attempting to add an NSOperation that is not an ASIHTTPRequest will generate an exception.

There is a complete example here that shows the basics of creating and using an ASINetworkQueue:

http://gist.github.com/150447

Cancelling an asynchronous request

To cancel an asynchronous request (either a request that was started with [request startAsynchronous] or a request running in a queue you created), call **[request cancel]**. Note that you cannot cancel a synchronous request.

Note that when you cancel a request, the request will treat that as an error, and will call your delegate and/or queue's failure delegate method. If you do not want this behaviour, set your delegate to nil before calling cancel, or use the **clearDelegatesAndCancel** method instead.

```
// Cancels an asynchronous request
[request cancel]

// Cancels an asynchronous request, clearing all delegates and blocks first
[request clearDelegatesAndCancel];
```

When using an ASINetworkQueue, all other requests will be cancelled when you cancel an individual request unless the queue's **shouldCancelAllRequestsOnFailure** is NO (YES is the default).

```
// When a request in this queue fails or is cancelled, other requests will continue to run
[queue setShouldCancelAllRequestsOnFailure:NO];

// Cancel all requests in a queue
[queue cancelAllOperations];
```

Safely handling the delegate being deallocated before the request has finished

Requests don't retain their delegates, so if there's a chance your delegate may be deallocated while your request is running, it is vital that you clear the request's delegate properties. In most circumstances, if your delegate is going to be deallocated, you probably also want to cancel request, since you no longer care about the request's status.

In the example below, our controller has an ASIHTTPRequest stored in a retained instance variable. We call the clearDelegatesAndCancel method in it's dealloc implementation, just before we release our reference to the request:

```
// Ddealloc method for our controller
- (void)dealloc
{
    [request clearDelegatesAndCancel];
    [request release];
    ...
    [super dealloc];
}
```

Sending data

Setting request headers

```
ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
[request addRequestHeader:@"Referer" value:@"http://allseeing-i.com/"];
```

Sending a form POST with ASIFormDataRequest

To send POST data in a manner compatible with web page forms, use the included **ASIFormDataRequest subclass**. Data is posted in 'application/x-www-form-urlencoded' format, or 'multipart/form-data' format when uploading binary data or files. Data in files is read as needed from disk, so POSTing large files is OK, as long as your web server is setup to handle them.

```
ASIFormDataRequest *request = [ASIFormDataRequest requestWithURL:url];
[request setPostValue:@"Ben" forKey:@"first_name"];
[request setPostValue:@"Copsey" forKey:@"last_name"];
[request setFile:@"/Users/ben/Desktop/ben.jpg" forKey:@"photo"];
```

ASIFormDataRequest will autodetect the mime type of files (on iOS 3.0 or later an Mac) added to the POST via setFile:forKey:, and include this in the mime headers sent to the server. If you prefer, you can use the longer form to override this:

```
ASIFormDataRequest *request = [ASIFormDataRequest requestWithURL:url];

// Upload a file on disk
[request setFile:@"/Users/ben/Desktop/ben.jpg" withFileName:@"myphoto.jpg" andContentType:@"image/jpeg"
forKey:@"photo"];

// Upload an NSData instance
[request setData:imageData withFileName:@"myphoto.jpg" andContentType:@"image/jpeg" forKey:@"photo"];
```

You can send multiple values for the same parameter using the alternative *add* API:

```
ASIFormDataRequest *request = [ASIFormDataRequest requestWithURL:url];
[request addPostValue:@"Ben" forKey:@"names"];
[request addPostValue:@"George" forKey:@"names"];
[request addFile:@"/Users/ben/Desktop/ben.jpg" forKey:@"photos"];
[request addData:imageData withFileName:@"george.jpg" andContentType:@"image/jpeg" forKey:@"photos"];
```

See **ASIFormDataRequest.h** for a full list of all the ways to add parameters to your POST.

PUT requests and custom POSTs

If you want to send data via PUT, or want to send it with POST but prefer to create the POST body yourself, use appendPostData: or appendPostDataFromFile:.

```
ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
[request appendPostData:[@"This is my data" dataUsingEncoding:NSUTF8StringEncoding]];
// Default becomes POST when you use appendPostData: / appendPostDataFromFile: / setPostBody:
[request setRequestMethod:@"PUT"];
```

If you want to send large amounts of data and aren't using an ASIFormDataRequest, see the **information on streaming post bodies from disk** below.

Downloading data

Downloading the response directly to a file

If the resource you are requesting is fairly large, you can save memory by downloading directly to a file. This way, ASIHTTPRequest won't have to keep the whole response in memory at once.

```
ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
[request setDownloadDestinationPath:@"/Users/ben/Desktop/my_file.txt"];
```

When downloading data to a file using **downloadDestinationPath**, data will be saved in a temporary file while the request is in progress. This file's path is stored in **temporaryFileDownloadPath**. When the request completes successfully, one of two things happen:

- If the data is gzip compressed (see **information on gzip** compression), the compressed file will be uncompressed into downloadDestinationPath, and the temporary file will be deleted
- If the data is not compressed, the temporary file is moved to downloadDestinationPath, overwriting any previous file

Note that if the response body is empty, no file will be created. If there is a possibility that a request may return an empty body, make sure you check the file exists before you attempt to do anything with it.

Handling the response data as it arrives

If you need to process the response as it comes in (for example, you want to use a streaming parser to parse the response while it is still being downloaded), have your delegate implement **request:didReceiveData:** (see **ASIHTTPRequestDelegate.h**). Note that when you do this, ASIHTTPRequest will not populate responseData or write the response to downloadDestinationPath - you must store the response yourself if you need to.

Reading the HTTP status code

ASIHTTPRequest doesn't do anything special with most HTTP status codes (except redirection and authentication status codes, see below for more info), so it's up to you to look out for problems (eg: 404) and make sure you act appropriately.

```
ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
[request startSynchronous];
int statusCode = [request responseStatusCode];
NSString *statusMessage = [request responseStatusMessage];
```

Reading response headers

```
ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
[request startSynchronous];
NSString *poweredBy = [[request responseHeaders] objectForKey:@"X-Powered-By"];
NSString *contentType = [[request responseHeaders] objectForKey:@"Content-Type"];
```

Handling text encodings

ASIHTTPRequest will attempt to read the text encoding of the received data from the Content-Type header. If it finds a text encoding, it will set responseEncoding to the appropriate **NSStringEncoding**. If it does not find a text encoding in the header, it will use the value of defaultResponseEncoding (this defaults to NSISOLatin1StringEncoding).

When you call [request responseString], ASIHTTPRequest will attempt to create a string from the data it received, using responseEncoding as the source encoding.

Handling redirection

ASIHTTPRequest will automatically redirect to a new URL when it encounters one of the following HTTP status codes, assuming a Location header was sent:

- 301 Moved Permanently
- 302 Found
- 303 See Other

When redirection occurs, the value of the response data (responseHeaders / responseCookies / responseData / responseString etc) will reflect the content received from the final location.

Cookies set on any of the urls encountered during a redirection cycle will be stored in the global cookie store, and will be represented to the server on the redirected request when appropriate.

You can turn off automatic redirection by setting the request's **shouldRedirect** property to NO.

By default, automatic redirects always redirect using a GET request (with no body). This behaviour matches the way most browsers work, not the HTTP spec, which specifies that 301 and 302 redirects should use the original method.

To preserve the original method (including the request body) for 301 and 302 redirects, set **shouldUseRFC2616RedirectBehaviour** to YES on the request before you start it.

Tracking progress

Each ASIHTTPRequest has two delegates that can be used for tracking progress - downloadProgressDelegate (for downloads) and uploadProgressDelegate (for uploads).

Progress delegates can be NSProgressIndicators (Mac OS X) or UIProgressViews (iPhone). ASIHTTPRequest will automatically cater for the differences in the behaviour of these two classes. If you want, you can also use a custom class as a progress delegate, as long as it responds to setProgress:.

- If you are performing a single request, you set an upload and/or download progress delegate on that request
- If you are performing multiple requests in a queue, and you want to track overall progress for all requests in the queue, use a ASINetworkQueue and set the progress delegate of the queue
- If you want to do both of these at the same time, this is possible too (as of v0.97)

IMPORTANT: If you are uploading to a site that requires authentication, the upload progress will be reset to its previous value every time it fails to supply valid authentication. For this reason, it is recommended that you only ever use an upload progress delegate when **useSessionPersistence** is YES (see below) when communicating with authenticating web servers, and ensure that you authenticate in another request before attempting to track the upload of a large amount of data.

Tracking progress for uploads where the request body is less than 128KB is currently not possible. For requests larger than 128KB, progress delegates will not receive information on the progress of the first 128KB of post data. This is because of limitations in the CFNetwork API. ~~I have filed a feature enhancement with Apple (bug id 6596016), hopefully they will modify CFNetwork in future to make this possible.~~

Update 21st June 2009: The wonderful folks at Apple were kind enough to address my bug report! In the iPhone 3.0 SDK, it looks like the buffer size has been reduced to 32KB, which makes accurate upload progress tracking a lot more reliable.

Tracking download progress for a single request

In this example, myProgressIndicator is an NSProgressIndicator.

```
ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
[request setDownloadProgressDelegate:myProgressIndicator];
[request startSynchronous];
NSLog(@"Max: %f, Value: %f", [myProgressIndicator maxValue],[myProgressIndicator doubleValue]);
```

Tracking download progress for a set of requests

In this example, myProgressIndicator is an UIProgressView, myQueue is an ASINetworkQueue.

```
- (void)fetchThisURLFiveTimes:(NSURL *)url
{
   [myQueue cancelAllOperations];
   [myQueue setDownloadProgressDelegate:myProgressIndicator];
   [myQueue setDelegate:self];
   [myQueue setRequestDidFinishSelector:@selector(queueComplete:)];
   int i;
   for (i=0; i<5; i++) {
      ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
      [myQueue addOperation:request];
   }
   [myQueue go];
}

- (void)queueComplete:(ASINetworkQueue *)queue
{
   NSLog(@"Value: %f", [myProgressIndicator progress]);
}
```

Notice that for ASINetworkQueues, we have to call [myQueue go] to start the queue.

Tracking upload progress for a single request

In this example, myProgressIndicator is an UIProgressView.

```
ASIFormDataRequest *request = [ASIFormDataRequest requestWithURL:url];
[request setPostValue:@"Ben" forKey:@"first_name"];
[request setPostValue:@"Copsey" forKey:@"last_name"];
[request setUploadProgressDelegate:myProgressIndicator];
[request startSynchronous];
NSLog(@"Value: %f",[myProgressIndicator progress]);
```

Tracking upload progress for a set of requests

In this example, myProgressIndicator is an NSProgressIndicator, myQueue is an ASINetworkQueue.

```
- (void)uploadSomethingFiveTimes:(NSURL *)url
{
   [myQueue cancelAllOperations];
   [myQueue setUploadProgressDelegate:myProgressIndicator];
   [myQueue setDelegate:self];
   [myQueue setRequestDidFinishSelector:@selector(queueComplete:)];
   int i;
```

```
    int i;
    for (i=0; i<5; i++) {
        ASIHTTPRequest *request = [ASIFormDataRequest requestWithURL:url];
        [request setPostBody:[@"Some data" dataUsingEncoding:NSUTF8StringEncoding]];
        [myQueue addOperation:request];
    }
    [myQueue go];
}

- (void)queueComplete:(ASINetworkQueue *)queue
{
    NSLog(@"Max: %f, Value: %f", [myProgressIndicator maxValue],[myProgressIndicator doubleValue]);
}
```

Accurate progress vs simple progress

ASIHTTPRequest provides two approaches for displaying progress, simple progress, and accurate progress. They are controlled using the showAccurateProgress of ASIHTTPRequests and ASINetworkQueues. When you set showAccurateProgress on a request, it affects only that request. When you set it on a queue, it affects all requests in the queue.

Simple progress

When using simple progress, progress will update only when a request completes. For a single request, this means you get two progress updates - 0% complete, and 100% complete. For a queue with four requests, you would get five progress updates, 0%, 25%, 50%, 75% and 100%, with each increment representing a request that has completed.

Simple progress (showAccurateProgress = NO) is the default for ASINetworkQueues, and is well suited to queues with a large number of small uploads / downloads.

Accurate progress

When using accurate progress, progress will be updated as bytes are sent or received. It is best for requests that send or receive a large amount of data, and will give a much better indication of how much data has been sent or received for requests that take some time.

Using accurate progress will slightly degrade the performance of uploads, since progress delegates (which are likely to be UIProgressViews or NSProgressIndicators) will be redrawn much more frequently.

Using accurate progress has a much greater effect on performance for downloads when using a queue, since the queue will first perform a HEAD request for each GET request it contains so it can determine the overall size of the data to be downloaded before downloading commences. Using accurate progress is highly recommended if you are downloading large files from a queue, but you should avoid it for queues that contain a large number of small downloads.

Accurate progress (showAccurateProgress == YES) is the default for ASIHTTPRequests that run synchronously.

Custom progress tracking

The ASIProgressDelegate protocol defines all the ways a delegate can get updates on the progress of a request. In most situations, setting your uploadProgressDelegate and/or downloadProgressDelegate to an NSProgressIndicator or UIProgressView will be sufficient. However, if you want to perform more complex progress tracking, your progress delegates should implement the following methods in preference to setProgress: (iOS) or setDoubleValue: / setMaxValue: (Mac). These methods allow you get updates on the actual number of bytes that are sent or received, rather than the number between 0 and 1 you get for the simpler methods.

Methods for downloadProgressDelegates

- **request:didReceiveBytes:** will be called on your downloadProgressDelegate each time the request downloads a bit more data. (Note that this is distinct from the **request:didReceiveData:** method that you regular delegate may implement).
- **request:incrementDownloadSizeBy:** will be called when the size of the download changes, with the passed parameter being the amount you should increase the size of the download by. This usually happens when the request receives response headers and finds out the size of the download.

Methods for uploadProgressDelegates

Methods for uploadProgressDelegates

- **request:didSendBytes:** will be called on your uploadProgressDelegate each time the request has been able to send a bit more data. **Important: This method will be called with a number less than zero** when a request needs to remove upload progress (usually when it has uploaded the data, but failed authentication or needs to run again for some other reason).
- **request:incrementUploadSizeBy:** will be called when the size of the upload changes. **The passed size will frequently be below zero**, as the request adjusts the upload size to take account of the size of the internal buffer used by the OS.

Handling HTTP authentication

If you're connecting to a server that requires authentication, you might want to take a look at **this flowchart** that shows how ASIHTTPRequest finds and applies credentials to requests.

Specifying a username and password to use in the URL

```
NSURL *url = [NSURL URLWithString:@"http://username:password@allseeing-i.com/top_secret/"];
ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
```

Setting a username and password to use on the request

```
NSURL *url = [NSURL URLWithString:@"http://allseeing-i.com/top_secret/"];
ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
[request setUsername:@"username"];
[request setPassword:@"password"];
```

Storing credentials in the keychain

If you turn on **keychainPersistence**, any valid username and password supplied will be stored in the keychain. Subsequent requests will reuse the username and password from the keychain, even if you quit and relaunch the application.

```
NSURL *url = [NSURL URLWithString:@"http://allseeing-i.com/top_secret/"];
ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
[request setUseKeychainPersistence:YES];
[request setUsername:@"username"];
[request setPassword:@"password"];
```

If you want to use the keychain but would rather manage it yourself, you might find the class methods relating to the keychain in ASIHTTPRequest.h helpful.

Storing credentials in the session

If **useSessionPersistence** is turned on (it is by default), ASIHTTPRequest stores credentials in memory and can re-use them for subsequent requests.

```
NSURL *url = [NSURL URLWithString:@"http://allseeing-i.com/top_secret/"];
ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
[request setUsername:@"username"];
[request setPassword:@"password"];
[request setUseSessionPersistence:YES]; //Shouldn't be needed as this is the default

//Should reuse our username and password
request = [ASIHTTPRequest requestWithURL:url];
```

NTLM authentication

To authenticate with a Windows server that uses the NTLM scheme, you also need to specify the domain you are

To authenticate with a Windows server that uses the NTLM scheme, you also need to specify the domain you are authenticating against.

```
NSURL *url = [NSURL URLWithString:@"http://my.windows.server/top_secret/"];
ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
[request setUsername:@"username"];
[request setPassword:@"password"];
[request setDomain:@"my-domain"];
```

Using delegation to provide credentials

Rather than specifying authentication credentials in advance, you might prefer to let each request ask its delegate for credentials if it can't find them in the session authentication cache or keychain. This might be helpful if you want to connect to a server where you aren't sure in advance what kind of authentication (if any) will be required.

Make sure your delegate implements authenticationNeededForRequest:, and ASIHTTPRequest will pause a request while it waits for the delegate to let it know what credentials to use. When you have the credentials you need, simply set them on the request, and call [request retryUsingSuppliedCredentials];. If you want to cancel, you must call [request cancelAuthentication]. This will cancel the request.

As of v1.0.8, request delegates will only receive one authenticationNeededForRequest: or proxyAuthenticationNeededForRequest: at once. Other requests requiring authentication will pause while the delegate handles the first request. If credentials are provided, any other requests currently in progress will attempt to reuse them, assuming they are valid for the url. If the delegate cancels authentication, and the queue's shouldCancelAllRequestsOnFailure is YES, all the other requests will cancel without attempting to ask for credentials.

You cannot use the delegation pattern for authentication when using synchronous requests.

In older versions this would have caused your app to hang, as of v1.0.8 the delegate methods will no longer be called.

Using the built-in authentication dialog (currently iOS only)

New in v1.0.8 is the ASIAuthenticationDialog class. This is primarily used for working with authenticating proxies (**see below**), but it can be used to ask the user for credentials for authenticating webservers.

For the best user experience, most apps that connect to a single service should implement authenticationNeededForRequest: in their request delegates, or avoid the use of delegation-style authentication altogether. However, there may be some occasions when the use of ASIHTTPRequest's standard authentication dialog for regular authentication may be advantageous:

- You don't want to create your own login form
- You may need to fetch data from an external source, and aren't certain if it will require authentication or not

For these cases, set shouldPresentAuthenticationDialog to true on your requests, and, if your delegate does not implement authenticationNeededForRequest:, users will see the dialog.

Only one dialog can appear at once, and other requests requiring authentication will pause while the dialog is visible. If credentials are provided, any other requests currently in progress will attempt to reuse them, assuming they are valid for the url. If the delegate cancels authentication, and the queue's shouldCancelAllRequestsOnFailure is YES (as per the default), all the other requests will cancel without attempting to ask for credentials.

The authentication dialog will not appear for synchronous requests.

The dialog is partly modelled on the authentication dialog that Safari uses on the iPhone, and includes:

- A message to indicate these credentials are for a webserver (as opposed to a proxy)

- A message to indicate these credentials are for a webserver (as opposed to a proxy)
- The hostname or IP of the server you are connecting to
- The authentication realm (if one was supplied)
- Fields to enter a username and password
- When connecting to a server that uses the NTLM scheme, the dialog also includes a field for the domain
- A note about whether the credentials are sent in plain text or not (ie: They are sent in plain text only when using Basic authentication without SSL)

If you want to change the look and feel, subclass ASIHTTPRequest, and override showAuthenticationDialog to show your custom dialog or ASIAuthenticationDialog subclass.

Presenting credentials before the server asks for them

IMPORTANT
Behaviour for this feature has changed in v1.8.1 for requests using Basic authentication, and you may need to update your code.

ASIHTTPRequest can present credentials to the server (if it has credentials to use) when it first makes a request, rather than waiting for the server to ask for credentials. This can result in better performance for applications using authentication, since it avoids an extra request.

To trigger this behaviour for requests using Basic authentication, you should manually set the request's authenticationScheme to use Basic:

```
[request setAuthenticationScheme:(NSString *)kCFHTTPAuthenticationSchemeBasic];
```

For other authentication schemes, credentials can be presented before the server asks for them, but only after another request has successfully authenticated with the server.

You may wish to disable this feature if:

- Your application may use multiple sets of credentials to talk to the same server at once
- Security is paramount for your application. Using this feature is inherently less secure, since credentials are sent before you have a chance to validate that you are connecting to the server you thought you were connecting to.

To disable this feature, use this code:

```
[request setShouldPresentCredentialsBeforeChallenge:NO];
```

Cookies

Persistent cookies

ASIHTTPRequest allows you to use the global store shared by all Mac OS X applications that use the CFNetwork or NSURLRequest APIs. If **useCookiePersistence** is on (it is by default), cookies will be stored in the shared NSHTTPCookieStorage container, and reused on other requests automatically. It's worth noting that ASIHTTPRequest might present cookies created in other applications if they are valid for a particular request.

You can clear all cookies created during a session like so:

```
[ASIHTTPRequest setSessionCookies:nil];
```

In this case, 'session cookies' refers to ALL cookies created during a session, rather cookies with no expiry date (often referred to as session cookies) that are removed when the application quits.

Alternatively, the convenience class method **clearSession** will clear all cookies created during the session, along with any cached authentication data.

Handling cookies yourself

If you prefer, you can turn off useCookiePersistence, and manage the set of cookies for a particular request manually:

```
//Create a cookie
NSDictionary *properties = [[[NSMutableDictionary alloc] init] autorelease];
[properties setValue:[@"Test Value" encodedCookieValue] forKey:NSHTTPCookieValue];
[properties setValue:@"ASIHTTPRequestTestCookie" forKey:NSHTTPCookieName];
[properties setValue:@".allseeing-i.com" forKey:NSHTTPCookieDomain];
[properties setValue:[NSDate dateWithTimeIntervalSinceNow:60*60] forKey:NSHTTPCookieExpires];
[properties setValue:@"/asi-http-request/tests" forKey:NSHTTPCookiePath];
NSHTTPCookie *cookie = [[[NSHTTPCookie alloc] initWithProperties:properties] autorelease];

//This url will return the value of the 'ASIHTTPRequestTestCookie' cookie
url = [NSURL URLWithString:@"http://allseeing-i.com/ASIHTTPRequest/tests/read_cookie"];
request = [ASIHTTPRequest requestWithURL:url];
[request setUseCookiePersistence:NO];
[request setRequestCookies:[NSMutableArray arrayWithObject:cookie]];
[request startSynchronous];

//Should be: I have 'Test Value' as the value of 'ASIHTTPRequestTestCookie'
NSLog(@"%@",[request responseString]);
```

Handling compressed responses, and compressing request bodies

Using gzip to handle compressed response data

As of v0.9, ASIHTTPRequest will notify the server that it can accept data compressed using gzip. If you are upgrading ASIHTTPRequest in an existing project, see the **setup instructions** for details of how to link with zlib.

Many webservers can compress data before sending it to the client - this results in faster downloads and lower bandwidth use, at the cost of additional CPU time on the server (to compress the data) and on the client (to decompress the data). Generally speaking, only certain types of data will be compressed - many binary formats like JPEG, GIF, PNG, SWF and PDF already compress their data, so gzip compression is normally not used to send them to a client. Text files like web pages and XML documents are perfect candidates for gzip compression, since they often contain a large amount of repeated information.

How to setup apache to gzip data using mod_deflate

Apache 2.x and above comes with the mod_deflate extension that allows it to transparently compress certain types of data. To turn it on, you need to enable mod_deflate in your apache configuration file, and add the mod_deflate directives to your virtual host config, or to your .htaccess file. **More information is available here**.

Using gzip in ASIHTTPRequest

```
- (IBAction)grabURL:(id)sender
{
  NSURL *url = [NSURL URLWithString:@"http://allseeing-i.com"];
  ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
  // YES is the default, you can turn off gzip compression by setting this to NO
  [request setAllowCompressedResponse:YES];
  [request startSynchronous];
  BOOL *dataWasCompressed = [request isResponseCompressed]; // Was the response gzip compressed?
  NSData *compressedResponse = [request rawResponseData]; // Compressed data
  NSData *uncompressedData = [request responseData]; // Uncompressed data
  NSString *response = [request responseString]; // Uncompressed data as a string
}
```

When allowCompressedResponse is true, ASIHTTPRequest will add a Accept Encoding header to the request

When allowCompressedResponse is true, ASIHTTPRequest will add a Accept-Encoding header to the request specifying we can accept gzipped response data. If the response headers contain a Content-Encoding header that specifies the data is compressed, calls to **responseData** or **responseString** will uncompress the data before returning it. You can get the original compressed data by calling **rawResponseData**.

On-the-fly inflating of gzipped responses

By default, ASIHTTPRequest will wait until a request finishes to inflate (uncompress) a gzipped response. By setting a request's **shouldWaitToInflateCompressedResponses** property to NO, you can tell ASIHTTPRequest to inflate the data as is comes in. In some circumstances, this may result in a small speed boost, as data can be processed while a request is waiting for more of the response.

This feature may be especially useful if you need to parse the response with a streaming parser (eg an XML or JSON parser). With this option enabled, you can feed inflated data directly to your parser as it comes in by implementing **request:didReceiveData:** in your delegate.

Note that when shouldWaitToInflateCompressedResponses is set to NO, the raw (compressed) data will be discarded. See the comments in ASIHTTPRequest.h for more info.

Using gzip to compress request bodies

New in v1.0.3 is gzip compression for request bodies. Using this feature, your applications can compress the content of POST / PUT operations by setting **shouldCompressRequestBody** to YES. shouldCompressRequestBody is NO by default.

Apache's mod_deflate can automatically inflate (decompress) gzipped request bodies when configured with *SetInputFilter DEFLATE* (**More info**). This approach works for CGI content, but not when you are using an Apache module built as a RESOURCE filter (such as mod PHP). In these cases, you need to inflate the data yourself.

ASIHTTPRequest cannot check if a server can accept a gzipped request body or not. Use this feature only if you are certain that the server you are talking to will understand a gzipped body.

Avoid gzipping content that is in a format that already uses compression (eg JPEG/PNG/GIF/PDF/SWF), you may well find the gzipped version is larger than the original.

Resuming interrupted downloads

As of v0.94, ASIHTTPRequest can resume partial downloads.

```
- (IBAction)resumeInterruptedDownload:(id)sender
{
  NSURL *url = [NSURL URLWithString:
    @"http://allseeing-i.com/ASIHTTPRequest/Tests/the_great_american_novel.txt"];
  ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];

  NSString *downloadPath = @"/Users/ben/Desktop/my_work_in_progress.txt";

  // The full file will be moved here if and when the request completes successfully
  [request setDownloadDestinationPath:downloadPath];

  // This file has part of the download in it already
  [request setTemporaryFileDownloadPath:@"/Users/ben/Desktop/my_work_in_progress.txt.download"];
  [request setAllowResumeForFileDownloads:YES];
  [request startSynchronous];

  //The whole file should be here now.
  NSString *theContent = [NSString stringWithContentsOfFile:downloadPath];
}
```

This only works for downloading data to a file, and you must set **allowResumeForFileDownloads** to YES for:

- Any download you might want to resume in future (or ASIHTTPRequest will remove the temporary download

- Any download you might want to resume in future (or ASIHTTPRequest will remove the temporary download file when it is cancelled or dealloced)
- Any download you want to resume

Additionally, you must set a temporary download path yourself (**setTemporaryFileDownloadPath**), with the path of the partial data. New data will be appended to this file, and the file moved to **downloadDestinationPath** when the download completes successfully.

Resuming works by reading the size of the file at temporaryFileDownloadPath, and then requesting the rest of the file using a **Range: bytes=x** HTTP header.

ASIHTTPRequest does not check for the presence of a **Accept-Ranges** header (because of the overhead of an additional HEAD request), so only use this feature when you are certain the that server you are connecting to supports partial downloads for the resource you want to download.

Streaming request bodies directly from disk

As of v0.96, ASIHTTPRequest can use files on disk as the request body. This means that it is no longer necessary to hold the request body in memory, which should result in a drastic reduction in memory usage for large POST/PUT operations.

There are several ways you can use this feature:

ASIFormDataRequests

```
NSURL *url = [NSURL URLWithString:@"http://allseeing-i.com/ignore"];
ASIFormDataRequest *request = [ASIFormDataRequest requestWithURL:url];
[request setPostValue:@"foo" forKey:@"post_var"];
[request setFile:@"/Users/ben/Desktop/bigfile.txt" forKey:@"file"];
[request startSynchronous];
```

ASIFormDataRequests automatically use this feature when you use setFile:forKey:. The request will create a temporary file that will contain the full post body. Files are written a bit at a time to the relevant part of the body. The request is created using CFReadStreamCreateForStreamedHTTPRequest, using a read stream on the file as the source.

Regular ASIHTTPRequests

If you know your request is going to be large, turn on streaming from disk on the request:

```
[request setShouldStreamPostDataFromDisk:YES];
```

In the example below, we add NSData objects to the post body one at a time. There are two methods for doing this - adding data from memory (appendPostData:), or appendPostDataFromFile: to add the contents of a file.

```
NSURL *url = [NSURL URLWithString:@"http://allseeing-i.com/ignore"];
ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
[request setShouldStreamPostDataFromDisk:YES];
[request appendPostData:myBigNSData];
[request appendPostDataFromFile:@"/Users/ben/Desktop/bigfile.txt"];
[request startSynchronous];
```

In this example, we want to PUT a large file directly. We set setPostBodyFilePath ourselves, ASIHTTPRequest will use this file as the post body.

```
NSURL *url = [NSURL URLWithString:@"http://allseeing-i.com/ignore"];
ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
[request setRequestMethod:@"PUT"];
```

```
[request setPostBodyFilePath:@"/Users/ben/Desktop/another-big-one.txt"];
[request setShouldStreamPostDataFromDisk:YES];
[request startSynchronous];
```

IMPORTANT: You should not use **setPostBody** in the same request as any of the methods described above - they are mutually exclusive. setPostBody should only be used if you want to build the request body yourself, and plan on keeping the request body in memory.

Using a download cache

The API for ASIDownloadCache and ASICacheDelegate has changed in v1.8, and you will need to update your code if you are updating from v1.7.
In particular, the options available for cache policies are different, and you can now combine multiple cache policies for a single request.

ASIHTTPRequest can automatically store downloaded data in a cache for use later. This can be helpful in many situations:

- You want to have access to the data when there is no internet connection and you can't download it again
- You want to download something only if it has changed since you last downloaded it
- The content you are working with will never change, so you only want to download it once

In previous versions of ASIHTTPRequest, handling the above situations would have meant storing the data manually yourself. Using a download cache may eliminate the need for writing any local storage mechanism yourself in some situations.

ASIDownloadCache is a simple URL cache that can be used for caching the response of GET requests. To be eligible for response caching, requests must succeed (no error), and the server must have returned a 200 OK HTTP response code, or, as of v1.8.1, a 301, 302, 303 and 307 redirect status code.

Turning on the response cache is easy:

```
[ASIHTTPRequest setDefaultCache:[ASIDownloadCache sharedCache]];
```

After you've done this, all requests will use the cache automatically. If you prefer, you can set individual requests to use the shared cache on a case by case basis:

```
ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
[request setDownloadCache:[ASIDownloadCache sharedCache]];
```

You aren't restricted to a single cache - you can create as many caches as you like. When you create a cache yourself, you must set the storage path for the cache - this should be a folder you have write access to:

```
ASIDownloadCache *cache = [[[ASIDownloadCache alloc] init] autorelease];
[cache setStoragePath:@"/Users/ben/Documents/Cached-Downloads"];

// Don't forget - you are responsible for retaining your cache!
[self setMyCache:cache];

ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
[request setDownloadCache:[self myCache]];
```

About cache policies

About cache policies

Cache policies are the main way you control when information is stored in the cache, and when cached data will be used in preference to downloading the data again.

The cache policy of individual requests can be controlled using a request's cachePolicy property. Cache policies are defined using a bitmask, so you can combine multiple options to create the policy you want:

```
// Always ask the server if there is new content available,
// If the request fails, use data from the cache even if it should have expired.
[request setCachePolicy:ASIAskServerIfModifiedCachePolicy|ASIFallbackToCacheIfLoadFailsCachePolicy];
```

You can use the following options to define a request's cache policy:

ASIUseDefaultCachePolicy	The default cache policy. When you set a request to use this, it will use the cache's defaultCachePolicy. ASIDownloadCache's default cache policy is 'ASIAskServerIfModifiedWhenStaleCachePolicy'. You should not combine this with other options.
ASIDoNotReadFromCacheCachePolicy	Requests will not read from the cache when using this option.
ASIDoNotWriteToCacheCachePolicy	Requests will not save to the cache when using this option.
ASIAskServerIfModifiedWhenStaleCachePolicy	This is the default cache policy for ASIDownloadCaches. When using this, requests will first look to see if a cached response is available in the cache. If there is no cached data, the request will proceed as normal. If there is and the cached data has not expired, the request will use the cached data without contacting the server. If the cached data *has* expired, the request will perform a conditional GET to see if an updated version is available. If the server says the cached data is current, cached data will be used, and new data will not be downloaded. In this case, the cache's expiry time will be updated to match the new expiry date from the server. If the server provided updated content, it will be downloaded, and the new data and expiry written to the cache.
ASIAskServerIfModifiedCachePolicy	This is the same as *ASIAskServerIfModifiedWhenStaleCachePolicy*, except that requests will always ask the server if updated data is available.
ASIOnlyLoadIfNotCachedCachePolicy	When using this option, cached data will always be used if it exists, even if it should have expired.
ASIDontLoadCachePolicy	When using this option, requests will succeed only if a response is already cached. If no response for a request is cached, the request will stop, and no error will be set on the request.
ASIFallbackToCacheIfLoadFailsCachePolicy	When using this option, requests will fallback to cached data if the request fails. If cached data is used after a failure, the request will succeed without error. You would normally use this option in combination with others, as it is only useful for specifying the behaviour to use when something goes wrong.

When you set the defaultCachePolicy property of a cache, all requests that use that cache will use that cache policy unless they have a custom cache policy set on themselves.

About storage polices

About storage policies

Storage policies allow you to define how long a cache will store a particular response. ASIHTTPRequest currently supports two storage policies:

ASICacheForSessionDurationCacheStoragePolicy is the default. Responses will be stored only for the duration of the session, and will be removed the first time the cache is used, or when [ASIHTTPRequest clearSession] is called.

With **ASICachePermanentlyCacheStoragePolicy**, cached responses are stored permanently. To use this storage policy, set it on a request:

```
ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
[request setCacheStoragePolicy:ASICachePermanentlyCacheStoragePolicy];
```

To manually clear the cache, call **clearCachedResponsesForStoragePolicy:**, passing the storage policy for the cached data you want to remove:

```
[[ASIDownloadCache sharedCache] clearCachedResponsesForStoragePolicy:ASICachePermanentlyCacheStoragePolicy];
```

Other cache-related features

```
// When you turn shouldRespectCacheControlHeaders off, the cache will store responses even if the server
// has explictly asked for them not be be cached (eg with a cache-control or pragma: no-cache header)
[[ASIDownloadCache sharedCache] setShouldRespectCacheControlHeaders:NO];

// Set secondsToCache on the request to override any expiry date for the content set by the server, and store
// this response in the cache until secondsToCache seconds have elapsed
ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
[request setSecondsToCache:60*60*24*30]; // Cache for 30 days

// After a request has run, didUseCachedResponse will return YES if the response was returned from the cache
[request didUseCachedResponse];

// Ask the cache for a path to store the response. This is the most efficient way to use a download cache,
// since data doesn't have to be copied into the cache when the request completes.
[request setDownloadDestinationPath:
   [[ASIDownloadCache sharedCache] pathToStoreCachedResponseDataForRequest:request]];
```

Writing your own cache

If you already have a download cache and would like to plug it in to ASIHTTPRequest, or you prefer to write your own, have your cache implement the **ASICacheDelegate** protocol.

Throttling bandwidth

As of v1.0.7, ASIHTTPRequest can throttle the bandwidth used by all requests to prevent it going over a user-defined limit. This may help iPhone applications that send or receive large amounts of data to make it through the app store review process.

Throttling works by using a global limit (in bytes) for how much data can be received or transmitted in one second. All requests share this limit. As they send or receive data, ASIHTTPRequest keeps track of how much data has been sent or received in the last second. If one request exceeds the limit, any others running will also have to wait for the remainder of the current measurement period.

On iOS, you can tell ASIHTTPRequest to automatically turn throttling on when using a WWAN (GPRS/Edge/3G) connection, and it will automatically turn it off when switching to WiFi.

```
// Will limit bandwidth to the predefined default for mobile applications when WWAN is active.

// Wi-Fi requests are not affected
```

```
// This method is only available on iOS
[ASIHTTPRequest setShouldThrottleBandwidthForWWAN:YES];

// Will throttle bandwidth based on a user-defined limit when when WWAN (not Wi-Fi) is active
// This method is only available on iOS
[ASIHTTPRequest throttleBandwidthForWWANUsingLimit:14800];

// Will prevent requests from using more than the predefined limit for mobile applications.
// Will limit ALL requests, regardless of whether Wi-Fi is in use or not - USE WITH CAUTION
[ASIHTTPRequest setMaxBandwidthPerSecond:ASIWWANBandwidthThrottleAmount];

// Log how many bytes have been received or sent per second (average from the last 5 seconds)
NSLog(@"%qi",[ASIHTTPRequest averageBandwidthUsedPerSecond]);
```

IMPORTANT: Read this before enabling bandwidth throttling:

- **Bandwidth throttling should be considered an experimental feature: Use at your own risk.**
- Do not set the bandwidth limit too low - it's probably best not to set it below ASIWWANBandwidthThrottleAmount
- The actual bandwidth used by your application will always be slightly more than the limit you have set, because the measured bandwidth does not include the bandwidth used by HTTP headers.
- The value of ASIWWANBandwidthThrottleAmount is not official, as far as I know, no bandwidth limit has been officially published.
- You should not turn on bandwidth throttling unless your application is likely to send or receive large amounts of data. It may be best to turn it only only while performing requests that download or upload a large amount of data, and leave it off at all other times.
- It probably goes without saying, but I make no guarantee your app won't be rejected for using excessive bandwidth when you turn on throttling.

Client certificates support

If your server requires the use of client certificates, as of v1.8 it is now possible to send them with your request.

```
// Will send the certificate attached to the identity (identity is a SecIdentityRef)
[request setClientCertificateIdentity:identity];

// Add an additional certificate (where cert is a SecCertificateRef)
[request setClientCertificates:[NSArray arrayWithObject:(id)cert]];
```

There is a helper function in **ClientCertificateTests.m** in the iPhone / iPad sample app that can create a SecIdentityRef from PKCS12 data (this function only works on iOS).

Working with Proxies

ASIHTTPRequest can detect system proxy settings and automatically apply them to requests. As of v1.0.6, it also supports PAC file proxy configuration, and authenticating proxies.

By default, ASIHTTPRequest will attempt to detect proxy settings automatically. However, should you wish, you can manually set proxy settings:

```
// Configure a proxy server manually
```

```
NSURL *url = [NSURL URLWithString:@"http://allseeing-i.com/ignore"];
ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
[request setProxyHost:@"192.168.0.1"];
[request setProxyPort:3128];

// Alternatively, you can use a manually-specified Proxy Auto Config file (PAC)
// (It's probably best if you use a local file)
[request setPACurl:[NSURL URLWithString:@"file:///Users/ben/Desktop/test.pac"]];
```

Authenticating proxies

On Mac OS, ASIHTTPRequest can auto-detect credentials used for authenticating proxies if they are specified in System Preferences. On iOS, ASIHTTPRequest cannot auto-detect the credentials used for authenticating proxies, so you either have to set them manually, use delegation to ask your controller / the user for appropriate credentials, or let **ASIAuthenticationDialog** ask the user for them. Once valid proxy credentials have been obtained, they are stored in the keychain (when useKeychainPersistence is on) and are automatically reused.

Manually specifying credentials for the proxy

```
NSURL *url = [NSURL URLWithString:@"http://allseeing-i.com/ignore"];
ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
[request setProxyHost:@"192.168.0.1"];
[request setProxyPort:3128];

// Set a username and password for authenticating proxies
[request setProxyUsername:@"bencopsey"];
[request setProxyPassword:@"password"];

// For NTLM proxies, you can also set the domain (NTLM proxies are untested!)
[request setProxyDomain:@"la.la.land"];
```

Using delegation to ask for proxy credentials

This works in the same way as **using delegates to provide credentials for regular authentication**, except that your delegate must respond to **proxyAuthenticationNeededForRequest:** (formerly proxyAuthorizationNeededForRequest:).

Using the built-in authentication dialog (currently iOS only)

New in v1.0.8 is the ASIAuthenticationDialog class. This can be used to ask the user for credentials for authenticating webservers and proxies.

If your delegate does not respond to proxyAuthenticationNeededForRequest:, by default, ASIHTTPRequest will show a dialog prompting the user to supply credentials. It appears by default for proxy servers so that all apps using ASIHTTPRequest can work with authenticating proxies without any additional effort on the part of the developer.

The proxy authentication dialog will not appear for synchronous requests.

If you prefer not to use the proxy authentication dialog, either implement proxyAuthenticationNeededForRequest: in your proxy, or set shouldPresentProxyAuthenticationDialog to false (in which case your application will not be able to connect to authenticating proxies). If you want to change the look and feel, subclass ASIHTTPRequest, and override showProxyAuthenticationDialog to show your custom dialog or ASIAuthenticationDialog subclass.

Miscellaneous features

Customising the user agent

To set the user agent your app will use, do this:

```
[ASIHTTPRequest setDefaultUserAgentString:@"MyApp 1.0"]
```

If you do not set a user agent, ASIHTTPRequest will create one for you. An example (for a Mac OS application):

My Application 1.0 (Macintosh; Mac OS X 10.5.7; en_GB)

You can also set the user agent on a per-request basis:

```
[request setUserAgent:@"MyApp 1.0"]
```

Continuing a request when your app enters the background on iOS

```
// iOS 4+ only
[request setShouldContinueWhenAppEntersBackground:YES];
```

Monitoring network activity

```
// Log the average bandwidth used (in bytes) per second over the last 5 seconds
NSLog(@"%llu",[ASIHTTPRequest averageBandwidthUsedPerSecond]);

if ([ASIHTTPRequest isNetworkInUse]) {
        // ASIHTTPRequest has requests in progress that are using the network
}
```

Disabling automatic updates to the network activity indicator (iOS only)

By default, ASIHTTPRequests will show the network activity indicator (in the status bar) on iOS devices when requests are using the network. If you prefer to manage this yourself, you can disable these updates:

```
[ASIHTTPRequest setShouldUpdateNetworkActivityIndicator:NO];
```

Automatically retry requests when they time out

Make requests retry a maximum of 2 times if they encounter a timeout:

```
[request setNumberOfTimesToRetryOnTimeout:2];
```

Configuring persistent connections

By default, ASIHTTPRequest will attempt to keep connections to a server open so that they can be reused by other requests to the same server (this generally results in significant speed boost, especially if you have many small requests). Persistent connections will be used automatically when connecting to an HTTP 1.1 server, or when the server sends a keep-alive header. Persistent connections are not used if the server explicitly sends a 'Connection: close' header. Additionally, ASIHTTPRequest will not use persistent connections for requests that include a body (eg POST/PUT) by default (as of v1.8.1). You can force the use of persistent connections for these request by manually setting the request method, then turning persistent connections back on:

```
[request setRequestMethod:@"PUT"];
[request setShouldAttemptPersistentConnection:YES];
```

Many servers do not provide any information in response headers on how long to keep a connection open, and may close the connection at any time after a request is finished. If the server does not send any information about how long the connection should be used, ASIHTTPRequest will keep connections to a server open for 60 seconds after any request has finished using them. Depending on your server configuration, this may be too long, or too short.

If this timeout is too long, the server may close the connection before the next request gets a chance to use it. When

ASIHTTPRequest encounters an error that appears to be a closed connection, it will retry the request on a new connection.

If this timeout is too short, and the server may be happy to keep the connection open for longer, but ASIHTTPRequest will needlessly open a new connection, which will incur a performance penalty.

```
// Set the amount of time to hang on to a persistent connection before it should expire to 2 minutes
[request setPersistentConnectionTimeoutSeconds:120];

// Disable persistent connections entirely
[request setShouldAttemptPersistentConnection:NO];
```

Forcing the use of HTTP 1.0

```
[request setUseHTTPVersionOne:YES];
```

Disabling secure certificate validation

You may wish to use this for testing purposes if you have a self-signed secure certificate. I recommend purchasing a certificate from a trusted certificate authority and leaving certificate validation turned on for production applications.

```
[request setValidatesSecureCertificate:NO];
```

Debugging options

ASIHTTPRequest provides a few flags that may be helpful in debugging request behaviour. These can be found in **ASIHTTPRequestConfig.h**.

When you turn on one of these flags, requests will print out information about what they are doing to the console.

DEBUG_REQUEST_STATUS

Prints information on the overall request lifecyle - starting, finishing upload, finishing download etc.

DEBUG_THROTTLING

Prints information on (roughly) how much bandwidth requests are using, and if requests are being throttled, information about how this is happening. In conjunction with DEBUG_REQUEST_STATUS, this can be helpful for debugging timeouts, as you can see the point at which a request stopped sending or receiving data.

DEBUG_PERSISTENT_CONNECTIONS

Prints information on how requests are re-using persistent connections. If you are seeing output like:

```
Request attempted to use connection #1, but it has been closed - will retry with a new connection
```

...this is an indication that the value you have for persistentConnectionTimeoutSeconds may be too high. See the **notes on configuring persistent connections** for more info.

DEBUG_HTTP_AUTHENTICATION

New in v1.8.1: This prints out information on how how request are handling HTTP authentication (Basic, Digest or NTLM).

DEBUG_FORM_DATA_REQUEST

Prints a summary of the request body that an ASIFormDataRequest will send. This is only helpful when using ASIFormDataRequest.