Life Cycle of a URL Session

You can use the `NSURLSession` API in two ways: with a system-provided delegate or with your own delegate. In general, you must use your own delegate if your app does any of the following:

- Uses background sessions to download or upload content while your app is not running.
- Performs custom authentication.
- Performs custom SSL certificate verification.
- Decides whether a transfer should be downloaded to disk or displayed based on the MIME type returned by the server or other similar criteria.
- Uploads data from a body stream (as opposed to an `NSData` object).
- Limits caching programmatically.
- Limits HTTP redirects programmatically.

If your app does not need to do any of these things, your app can use the system-provided delegates. Depending on which technique you choose, you should read one of the following sections:

- Life Cycle of a URL Session with System-Provided Delegates provides a lightweight view of how your code creates and uses a URL session. You should read this section even if you intend to write your own delegate, because it gives you a complete picture of what your code must do to configure the object and use it.
- Life Cycle of a URL Session with Custom Delegates provides a complete view of every step in the operation of a URL session. You should refer to this section to help you understand how the session interacts with its delegate. In particular, this explains when each of the delegate methods is called.

Life Cycle of a URL Session with System-Provided Delegates

If you are using the `NSURLSession` class without providing a delegate object, the system-provided delegate handles many of the details for you. Here is the basic sequence of method calls that your app must make and completion handler calls that your app receives when using `NSURLSession` with the system-provided delegate:

1. Create a session configuration. For background sessions, this configuration must contain a unique identifier. Store that identifier, and use it to reassociate with the session if your app crashes or is terminated or suspended.
2. Create a session, specifying a configuration object and a `nil` delegate.
3. Create task objects within a session that each represent a resource request.

   Each task starts out in a suspended state. After your app calls `resume` on the task, it begins downloading the specified resource.

   The task objects are subclasses of `NSURLSessionTask`—`NSURLSessionDataTask`, `NSURLSessionUploadTask`, or `NSURLSessionDownloadTask`, depending on the behavior you are trying to achieve. These objects are analogous to `NSURLConnection` objects, but give you more control and a unified delegate model.

   Although your app can (and typically should) add more than one task to a session, for simplicity, the remaining steps describe the life cycle in terms of a single task.

> **Important:** If you are using the `NSURLSession` class without providing delegates, your app must create tasks using a call that takes a `completionHandler` parameter, because otherwise it cannot obtain data from the class.

4. For a download task, during the transfer from the server, if the user tells your app to pause the download, cancel the task by calling `cancelByProducingResumeData:` method. Later, pass the returned resume data to either the `downloadTaskWithResumeData:` or `downloadTaskWithResumeData:completionHandler:` method to create a new download task that continues the download.
5. When a task completes, the `NSURLSession` object calls the task's completion handler.

> **Note:** `NSURLSession` does not report server errors through the error parameter. The only errors your app receives through the error parameter are client-side errors, such as being unable to resolve the hostname or connect to the host. The error codes are described in `URL Loading System Error Codes`.
>
> Server-side errors are reported through the HTTP status code in the `NSHTTPURLResponse` object. For more information, read the documentation for the `NSHTTPURLResponse` and `NSURLResponse` classes.

6. When your app no longer needs a session, invalidate it by calling either `invalidateAndCancel` (to cancel outstanding tasks) or `finishTasksAndInvalidate` (to allow outstanding tasks to finish before invalidating the object).

Life Cycle of a URL Session with Custom Delegates

You can often use the `NSURLSession` API without providing a delegate. However, if you are using the `NSURLSession` API for background downloads and uploads, or if you need to handle authentication or caching in a nondefault manner, you must provide a delegate that conforms to the session delegate protocol, one or more task delegate protocols, or some combination of these protocols. This delegate serves many purposes:

- When used with download tasks, the `NSURLSession` object uses the delegate to provide your app with a file URL where it can obtain the downloaded data.

  Delegates are required for all background downloads and uploads. These delegates must provide all of the delegate methods in the `NSURLSessionDownloadDelegate` protocol.
- Delegates can handle certain authentication challenges.
- Delegates provide body streams for uploading stream-based data to the remote server.
- Delegates can decide whether to follow HTTP redirects or not.
- The `NSURLSession` object uses the delegate to provide your app with the status of each transfer. Data task delegates receive both an initial call, in which you can convert the request into a download, and subsequent calls, which provide pieces of data as they arrive from the remote server.
- Delegates are one way in which the `NSURLSession` object can tell your app when a transfer is complete.

If you are using custom delegates with a URL session (required for background tasks), the complete life cycle of a URL session is more complex. Here is the basic sequence of method calls that your app must make and delegate calls that your app receives when using `NSURLSession` with a custom delegate:

1. Create a session configuration. For background sessions, this configuration must contain a unique identifier. Store that identifier, and use it to reassociate with the session if your app crashes or is terminated or suspended.
2. Create a session, specifying a configuration object and, optionally, a delegate.
3. Create task objects within a session that each represent a resource request.

   Each task starts out in a suspended state. After your app calls `resume` on the task, it begins downloading the specified resource.

   The task objects are subclasses of `NSURLSessionTask`—`NSURLSessionDataTask`, `NSURLSessionUploadTask`, or `NSURLSessionDownloadTask`, depending on the behavior you

are trying to achieve. These objects are analogous to `NSURLConnection` objects, but give you more control and a unified delegate model.

Although your app can (and typically should) add more than one task to a session, for simplicity, the remaining steps describe the life cycle in terms of a single task.

4. If the remote server returns a status code that indicates authentication is required and if that authentication requires a connection-level challenge (such as an SSL client certificate), `NSURLSession` calls an authentication challenge delegate method.
   - For session-level challenges—`NSURLAuthenticationMethodNTLM`, `NSURLAuthenticationMethodNegotiate`, `NSURLAuthenticationMethodClientCertificate`, or `NSURLAuthenticationMethodServerTrust`—the `NSURLSession` object calls the session delegate's `URLSession:didReceiveChallenge:completionHandler:` method. If your app does not provide a session delegate method, the `NSURLSession` object calls the task delegate's `URLSession:task:didReceiveChallenge:completionHandler:` method to handle the challenge.
   - For non-session-level challenges (all others), the `NSURLSession` object calls the session delegate's `URLSession:task:didReceiveChallenge:completionHandler:` method to handle the challenge. If your app provides a session delegate and you need to handle authentication, then you must either handle the authentication at the task level or provide a task-level handler that calls the per-session handler explicitly. The session delegate's `URLSession:didReceiveChallenge:completionHandler:` method is *not* called for non-session-level challenges.

   Note: Kerberos authentication is handled transparently.

   If authentication fails for an upload task, if the task's data is provided from a stream, the `NSURLSession` object calls the delegate's `URLSession:task:needNewBodyStream:` delegate method. The delegate must then provide a new `NSInputStream` object to provide the body data for the new request.

   For more information about writing an authentication delegate method for `NSURLSession`, read Authentication Challenges and TLS Chain Validation.

5. Upon receiving an HTTP redirect response, the `NSURLSession` object calls the delegate's `URLSession:task:willPerformHTTPRedirection:newRequest:completionHandler:` method. That delegate method calls the provided completion handler with either the provided `NSURLRequest` object (to follow the redirect), a new `NSURLRequest` object (to redirect to a different URL), or `nil` (to treat the redirect's response body as a valid response and return it as the result).
   - If the redirect is followed, go back to step 4 (authentication challenge handling).
   - If the delegate does not implement this method, the redirect is followed up to the maximum number of redirects.

6. For a (re-)download task created by calling `downloadTaskWithResumeData:` or `downloadTaskWithResumeData:completionHandler:`, `NSURLSession` calls the delegate's `URLSession:downloadTask:didResumeAtOffset:expectedTotalBytes:` method with the new task object.
7. For a data task, the `NSURLSession` object calls the delegate's `URLSession:dataTask:didReceiveResponse:completionHandler:` method. Decide whether to convert the data task into a download task, and then call the completion callback to continue receiving data or downloading data.

   If your app chooses to convert the data task to a download task, `NSURLSession` calls the delegate's `URLSession:dataTask:didBecomeDownloadTask:` method with the new download task as a parameter. After this call, the delegate receives no further callbacks from the data task, and begins receiving callbacks from the download task.

8. If the task was created with `uploadTaskWithStreamedRequest:`, `NSURLSession` calls the delegate's `URLSession:task:needNewBodyStream:` method to provide the body data.
9. During the initial upload of body content to the server (if applicable), the delegate periodically receives

`URLSession:task:didSendBodyData:totalBytesSent:totalBytesExpectedToSend:` callbacks that report the progress of the upload.

10. During the transfer from the server, the task delegate periodically receives a callback to report the progress of the transfer. For a download task, the session calls the delegate's `URLSession:downloadTask:didWriteData:totalBytesWritten:totalBytesExpectedToWrite:` method with the number of bytes successfully written to disk. For a data task, the session calls the delegate's `URLSession:dataTask:didReceiveData:` method with the actual pieces of data as they are received.

    For a download task, during the transfer from the server, if the user tells your app to pause the download, cancel the task by calling the `cancelByProducingResumeData:` method.

    Later, if the user asks your app to resume the download, pass the returned resume data to either the `downloadTaskWithResumeData:` or `downloadTaskWithResumeData:completionHandler:` method to create a new download task that continues the download, then go to step 3 (creating and resuming task objects).

11. For a data task, the `NSURLSession` object calls the delegate's `URLSession:dataTask:willCacheResponse:completionHandler:` method. Your app should then decide whether to allow caching. If you do not implement this method, the default behavior is to use the caching policy specified in the session's configuration object.

12. If a download task completes successfully, then the `NSURLSession` object calls the task's `URLSession:downloadTask:didFinishDownloadingToURL:` method with the location of a *temporary* file. Your app must either read the response data from this file or move it to a permanent location in your app's sandbox container directory before this delegate method returns.

13. When any task completes, the `NSURLSession` object calls the delegate's `URLSession:task:didCompleteWithError:` method with either an error object or `nil` (if the task completed successfully).

    If the task failed, most apps should retry the request until either the user cancels the download or the server returns an error indicating that the request will never succeed. Your app should not retry immediately, however. Instead, it should use reachability APIs to determine whether the server is reachable, and should make a new request only when it receives a notification that reachability has changed.

    If the download task can be resumed, the `NSError` object's `userInfo` dictionary contains a value for the `NSURLSessionDownloadTaskResumeData` key. Your app should pass this value to call `downloadTaskWithResumeData:` or `downloadTaskWithResumeData:completionHandler:` to create a new download task that continues the existing download.

    If the task cannot be resumed, your app should create a new download task and restart the transaction from the beginning.

    In either case, if the transfer failed for any reason other than a server error, go to step 3 (creating and resuming task objects).

    > **Note:** `NSURLSession` does not report server errors through the error parameter. The only errors your delegate receives through the error parameter are client-side errors, such as being unable to resolve the hostname or connect to the host. The error codes are described in `URL Loading System Error Codes`.
    >
    > Server-side errors are reported through the HTTP status code in the `NSHTTPURLResponse` object. For more information, read the documentation for the `NSHTTPURLResponse` and `NSURLResponse` classes.

14. If the response is multipart encoded, the session may call the delegate's `didReceiveResponse` method again, followed by zero or more additional `didReceiveData` calls. If this happens, go to step 7 (handling the `didReceiveResponse` call).

15. When you no longer need a session, invalidate it by calling either `invalidateAndCancel` (to cancel outstanding tasks) or `finishTasksAndInvalidate` (to allow outstanding tasks to finish before invalidating the object).

After invalidating the session, when all outstanding tasks have been canceled or have finished, the session sends the delegate a `URLSession:didBecomeInvalidWithError:` message. When that delegate method returns, the session disposes of its strong reference to the delegate.

> **Important:** The session object keeps a strong reference to the delegate until your app explicitly invalidates the session. If you do not invalidate the session, your app leaks memory.

If your app cancels an in-progress download, the `NSURLSession` object calls the delegate's `URLSession:task:didCompleteWithError:` method as though an error occurred.
