// Alamofire.swift

//

// Copyright (c) 2014 Alamofire (<http://alamofire.org)>

//

// Permission is hereby granted, free of charge, to any person obtaining a copy

// of this software and associated documentation files (the "Software"), to deal

// in the Software without restriction, including without limitation the rights

// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell

// copies of the Software, and to permit persons to whom the Software is

// furnished to do so, subject to the following conditions:

//

// The above copyright notice and this permission notice shall be included in

// all copies or substantial portions of the Software.

//

// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR

// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,

// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE

// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER

// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,

// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN

// THE SOFTWARE.

import Foundation

/// Alamofire errors

public let AlamofireErrorDomain = "com.alamofire.error"

/\*\*

HTTP method definitions.

See <http://tools.ietf.org/html/rfc7231#section-4.3>

\*/

public enum Method: String {

case OPTIONS = "OPTIONS"

case GET = "GET"

case HEAD = "HEAD"

case POST = "POST"

case PUT = "PUT"

case PATCH = "PATCH"

case DELETE = "DELETE"

case TRACE = "TRACE"

case CONNECT = "CONNECT"

}

/\*\*

Used to specify the way in which a set of parameters are applied to a URL request.

\*/

public enum ParameterEncoding {

/\*\*

A query string to be set as or appended to any existing URL query for `GET`, `HEAD`, and `DELETE` requests, or set as the body for requests with any other HTTP method. The `Content-Type` HTTP header field of an encoded request with HTTP body is set to `application/x-www-form-urlencoded`. Since there is no published specification for how to encode collection types, the convention of appending `[]` to the key for array values (`foo[]=1&foo[]=2`), and appending the key surrounded by square brackets for nested dictionary values (`foo[bar]=baz`).

\*/

case URL

/\*\*

Uses `NSJSONSerialization` to create a JSON representation of the parameters object, which is set as the body of the request. The `Content-Type` HTTP header field of an encoded request is set to `application/json`.

\*/

case JSON

/\*\*

Uses `NSPropertyListSerialization` to create a plist representation of the parameters object, according to the associated format and write options values, which is set as the body of the request. The `Content-Type` HTTP header field of an encoded request is set to `application/x-plist`.

\*/

case PropertyList(NSPropertyListFormat, NSPropertyListWriteOptions)

/\*\*

Uses the associated closure value to construct a new request given an existing request and parameters.

\*/

case Custom((URLRequestConvertible, [String: AnyObject]?) -> (NSURLRequest, NSError?))

/\*\*

Creates a URL request by encoding parameters and applying them onto an existing request.

:param: URLRequest The request to have parameters applied

:param: parameters The parameters to apply

:returns: A tuple containing the constructed request and the error that occurred during parameter encoding, if any.

\*/

public func encode(URLRequest: URLRequestConvertible, parameters: [String: AnyObject]?) -> (NSURLRequest, NSError?) {

if parameters == nil {

return (URLRequest.URLRequest, nil)

}

var mutableURLRequest: NSMutableURLRequest! = URLRequest.URLRequest.mutableCopy() as NSMutableURLRequest

var error: NSError? = nil

switch self {

case .URL:

func query(parameters: [String: AnyObject]) -> String {

var components: [(String, String)] = []

for key in sorted(Array(parameters.keys), <) {

let value: AnyObject! = parameters[key]

components += queryComponents(key, value)

}

return join("&", components.map{"\($0)=\($1)"} as [String])

}

func encodesParametersInURL(method: Method) -> Bool {

switch method {

case .GET, .HEAD, .DELETE:

return true

default:

return false

}

}

let method = Method(rawValue: mutableURLRequest.HTTPMethod)

if method != nil && encodesParametersInURL(method!) {

if let URLComponents = NSURLComponents(URL: mutableURLRequest.URL!, resolvingAgainstBaseURL: false) {

URLComponents.percentEncodedQuery = (URLComponents.query != nil ? URLComponents.query! + "&" : "") + query(parameters!)

mutableURLRequest.URL = URLComponents.URL

}

} else {

if mutableURLRequest.valueForHTTPHeaderField("Content-Type") == nil {

mutableURLRequest.setValue("application/x-www-form-urlencoded", forHTTPHeaderField: "Content-Type")

}

mutableURLRequest.HTTPBody = query(parameters!).dataUsingEncoding(NSUTF8StringEncoding, allowLossyConversion: false)

}

case .JSON:

let options = NSJSONWritingOptions.allZeros

if let data = NSJSONSerialization.dataWithJSONObject(parameters!, options: options, error: &error) {

mutableURLRequest.setValue("application/json", forHTTPHeaderField: "Content-Type")

mutableURLRequest.HTTPBody = data

}

case .PropertyList(let (format, options)):

if let data = NSPropertyListSerialization.dataWithPropertyList(parameters!, format: format, options: options, error: &error) {

mutableURLRequest.setValue("application/x-plist", forHTTPHeaderField: "Content-Type")

mutableURLRequest.HTTPBody = data

}

case .Custom(let closure):

return closure(mutableURLRequest, parameters)

}

return (mutableURLRequest, error)

}

func queryComponents(key: String, \_ value: AnyObject) -> [(String, String)] {

var components: [(String, String)] = []

if let dictionary = value as? [String: AnyObject] {

for (nestedKey, value) in dictionary {

components += queryComponents("\(key)[\(nestedKey)]", value)

}

} else if let array = value as? [AnyObject] {

for value in array {

components += queryComponents("\(key)[]", value)

}

} else {

components.extend([(escape(key), escape("\(value)"))])

}

return components

}

func escape(string: String) -> String {

let allowedCharacters = NSCharacterSet(charactersInString:" =\"#%/<>?@\\^`{}[]|&+").invertedSet

return string.stringByAddingPercentEncodingWithAllowedCharacters(allowedCharacters) ?? string

}

}

// MARK: - URLStringConvertible

/\*\*

Types adopting the `URLStringConvertible` protocol can be used to construct URL strings, which are then used to construct URL requests.

\*/

public protocol URLStringConvertible {

/// The URL string.

var URLString: String { get }

}

extension String: URLStringConvertible {

public var URLString: String {

return self

}

}

extension NSURL: URLStringConvertible {

public var URLString: String {

return absoluteString!

}

}

extension NSURLComponents: URLStringConvertible {

public var URLString: String {

return URL!.URLString

}

}

extension NSURLRequest: URLStringConvertible {

public var URLString: String {

return URL.URLString

}

}

// MARK: - URLRequestConvertible

/\*\*

Types adopting the `URLRequestConvertible` protocol can be used to construct URL requests.

\*/

public protocol URLRequestConvertible {

/// The URL request.

var URLRequest: NSURLRequest { get }

}

extension NSURLRequest: URLRequestConvertible {

public var URLRequest: NSURLRequest {

return self

}

}

// MARK: -

/\*\*

Responsible for creating and managing `Request` objects, as well as their underlying `NSURLSession`.

\*/

public class Manager {

/\*\*

A shared instance of `Manager`, used by top-level Alamofire request methods, and suitable for use directly for any ad hoc requests.

\*/

public class var sharedInstance: Manager {

struct Singleton {

static var configuration: NSURLSessionConfiguration = {

var configuration = NSURLSessionConfiguration.defaultSessionConfiguration()

configuration.HTTPAdditionalHeaders = {

// Accept-Encoding HTTP Header; see <http://tools.ietf.org/html/rfc7230#section-4.2.3>

let acceptEncoding: String = "gzip;q=1.0,compress;q=0.5"

// Accept-Language HTTP Header; see <http://tools.ietf.org/html/rfc7231#section-5.3.5>

let acceptLanguage: String = {

var components: [String] = []

for (index, languageCode) in enumerate(NSLocale.preferredLanguages() as [String]) {

let q = 1.0 - (Double(index) \* 0.1)

components.append("\(languageCode);q=\(q)")

if q <= 0.5 {

break

}

}

return join(",", components)

}()

// User-Agent Header; see <http://tools.ietf.org/html/rfc7231#section-5.5.3>

let userAgent: String = {

if let info = NSBundle.mainBundle().infoDictionary {

let executable: AnyObject = info[kCFBundleExecutableKey] ?? "Unknown"

let bundle: AnyObject = info[kCFBundleIdentifierKey] ?? "Unknown"

let version: AnyObject = info[kCFBundleVersionKey] ?? "Unknown"

let os: AnyObject = NSProcessInfo.processInfo().operatingSystemVersionString ?? "Unknown"

var mutableUserAgent = NSMutableString(string: "\(executable)/\(bundle) (\(version); OS \(os))") as CFMutableString

let transform = NSString(string: "Any-Latin; Latin-ASCII; [:^ASCII:] Remove") as CFString

if CFStringTransform(mutableUserAgent, nil, transform, 0) == 1 {

return mutableUserAgent as NSString

}

}

return "Alamofire"

}()

return ["Accept-Encoding": acceptEncoding,

"Accept-Language": acceptLanguage,

"User-Agent": userAgent]

}()

return configuration

}()

static let instance = Manager(configuration: configuration)

}

return Singleton.instance

}

private let delegate: SessionDelegate

private let queue = dispatch\_queue\_create(nil, DISPATCH\_QUEUE\_SERIAL)

/// The underlying session.

public let session: NSURLSession

/// Whether to start requests immediately after being constructed. `true` by default.

public var startRequestsImmediately: Bool = true

/\*\*

:param: configuration The configuration used to construct the managed session.

\*/

required public init(configuration: NSURLSessionConfiguration? = nil) {

self.delegate = SessionDelegate()

self.session = NSURLSession(configuration: configuration, delegate: delegate, delegateQueue: nil)

}

deinit {

self.session.invalidateAndCancel()

}

// MARK: -

/\*\*

Creates a request for the specified method, URL string, parameters, and parameter encoding.

:param: method The HTTP method.

:param: URLString The URL string.

:param: parameters The parameters. `nil` by default.

:param: encoding The parameter encoding. `.URL` by default.

:returns: The created request.

\*/

public func request(method: Method, \_ URLString: URLStringConvertible, parameters: [String: AnyObject]? = nil, encoding: ParameterEncoding = .URL) -> Request {

return request(encoding.encode(URLRequest(method, URLString), parameters: parameters).0)

}

/\*\*

Creates a request for the specified URL request.

If `startRequestsImmediately` is `true`, the request will have `resume()` called before being returned.

:param: URLRequest The URL request

:returns: The created request.

\*/

public func request(URLRequest: URLRequestConvertible) -> Request {

var dataTask: NSURLSessionDataTask?

dispatch\_sync(queue) {

dataTask = self.session.dataTaskWithRequest(URLRequest.URLRequest)

}

let request = Request(session: session, task: dataTask!)

delegate[request.delegate.task] = request.delegate

if startRequestsImmediately {

request.resume()

}

return request

}

class SessionDelegate: NSObject, NSURLSessionDelegate, NSURLSessionTaskDelegate, NSURLSessionDataDelegate, NSURLSessionDownloadDelegate {

private var subdelegates: [Int: Request.TaskDelegate]

private let subdelegateQueue = dispatch\_queue\_create(nil, DISPATCH\_QUEUE\_CONCURRENT)

private subscript(task: NSURLSessionTask) -> Request.TaskDelegate? {

get {

var subdelegate: Request.TaskDelegate?

dispatch\_sync(subdelegateQueue) {

subdelegate = self.subdelegates[task.taskIdentifier]

}

return subdelegate

}

set {

dispatch\_barrier\_async(subdelegateQueue) {

self.subdelegates[task.taskIdentifier] = newValue

}

}

}

var sessionDidBecomeInvalidWithError: ((NSURLSession!, NSError!) -> Void)?

var sessionDidFinishEventsForBackgroundURLSession: ((NSURLSession!) -> Void)?

var sessionDidReceiveChallenge: ((NSURLSession!, NSURLAuthenticationChallenge) -> (NSURLSessionAuthChallengeDisposition, NSURLCredential!))?

var taskWillPerformHTTPRedirection: ((NSURLSession!, NSURLSessionTask!, NSHTTPURLResponse!, NSURLRequest!) -> (NSURLRequest!))?

var taskDidReceiveChallenge: ((NSURLSession!, NSURLSessionTask!, NSURLAuthenticationChallenge) -> (NSURLSessionAuthChallengeDisposition, NSURLCredential?))?

var taskDidSendBodyData: ((NSURLSession!, NSURLSessionTask!, Int64, Int64, Int64) -> Void)?

var taskNeedNewBodyStream: ((NSURLSession!, NSURLSessionTask!) -> (NSInputStream!))?

var dataTaskDidReceiveResponse: ((NSURLSession!, NSURLSessionDataTask!, NSURLResponse!) -> (NSURLSessionResponseDisposition))?

var dataTaskDidBecomeDownloadTask: ((NSURLSession!, NSURLSessionDataTask!) -> Void)?

var dataTaskDidReceiveData: ((NSURLSession!, NSURLSessionDataTask!, NSData!) -> Void)?

var dataTaskWillCacheResponse: ((NSURLSession!, NSURLSessionDataTask!, NSCachedURLResponse!) -> (NSCachedURLResponse))?

var downloadTaskDidFinishDownloadingToURL: ((NSURLSession!, NSURLSessionDownloadTask!, NSURL) -> (NSURL))?

var downloadTaskDidWriteData: ((NSURLSession!, NSURLSessionDownloadTask!, Int64, Int64, Int64) -> Void)?

var downloadTaskDidResumeAtOffset: ((NSURLSession!, NSURLSessionDownloadTask!, Int64, Int64) -> Void)?

required override init() {

self.subdelegates = Dictionary()

super.init()

}

// MARK: NSURLSessionDelegate

func URLSession(session: NSURLSession!, didBecomeInvalidWithError error: NSError!) {

sessionDidBecomeInvalidWithError?(session, error)

}

func URLSession(session: NSURLSession!, didReceiveChallenge challenge: NSURLAuthenticationChallenge!, completionHandler: ((NSURLSessionAuthChallengeDisposition, NSURLCredential!) -> Void)!) {

if sessionDidReceiveChallenge != nil {

completionHandler(sessionDidReceiveChallenge!(session, challenge))

} else {

completionHandler(.PerformDefaultHandling, nil)

}

}

func URLSessionDidFinishEventsForBackgroundURLSession(session: NSURLSession!) {

sessionDidFinishEventsForBackgroundURLSession?(session)

}

// MARK: NSURLSessionTaskDelegate

func URLSession(session: NSURLSession!, task: NSURLSessionTask!, willPerformHTTPRedirection response: NSHTTPURLResponse!, newRequest request: NSURLRequest!, completionHandler: ((NSURLRequest!) -> Void)!) {

var redirectRequest = request

if taskWillPerformHTTPRedirection != nil {

redirectRequest = taskWillPerformHTTPRedirection!(session, task, response, request)

}

completionHandler(redirectRequest)

}

func URLSession(session: NSURLSession!, task: NSURLSessionTask!, didReceiveChallenge challenge: NSURLAuthenticationChallenge!, completionHandler: ((NSURLSessionAuthChallengeDisposition, NSURLCredential!) -> Void)!) {

if let delegate = self[task] {

delegate.URLSession(session, task: task, didReceiveChallenge: challenge, completionHandler: completionHandler)

} else {

URLSession(session, didReceiveChallenge: challenge, completionHandler: completionHandler)

}

}

func URLSession(session: NSURLSession!, task: NSURLSessionTask!, needNewBodyStream completionHandler: ((NSInputStream!) -> Void)!) {

if let delegate = self[task] {

delegate.URLSession(session, task: task, needNewBodyStream: completionHandler)

}

}

func URLSession(session: NSURLSession!, task: NSURLSessionTask!, didSendBodyData bytesSent: Int64, totalBytesSent: Int64, totalBytesExpectedToSend: Int64) {

if let delegate = self[task] as? Request.UploadTaskDelegate {

delegate.URLSession(session, task: task, didSendBodyData: bytesSent, totalBytesSent: totalBytesSent, totalBytesExpectedToSend: totalBytesExpectedToSend)

}

}

func URLSession(session: NSURLSession!, task: NSURLSessionTask!, didCompleteWithError error: NSError!) {

if let delegate = self[task] {

delegate.URLSession(session, task: task, didCompleteWithError: error)

self[task] = nil

}

}

// MARK: NSURLSessionDataDelegate

func URLSession(session: NSURLSession!, dataTask: NSURLSessionDataTask!, didReceiveResponse response: NSURLResponse!, completionHandler: ((NSURLSessionResponseDisposition) -> Void)!) {

var disposition: NSURLSessionResponseDisposition = .Allow

if dataTaskDidReceiveResponse != nil {

disposition = dataTaskDidReceiveResponse!(session, dataTask, response)

}

completionHandler(disposition)

}

func URLSession(session: NSURLSession!, dataTask: NSURLSessionDataTask!, didBecomeDownloadTask downloadTask: NSURLSessionDownloadTask!) {

let downloadDelegate = Request.DownloadTaskDelegate(task: downloadTask)

self[downloadTask] = downloadDelegate

}

func URLSession(session: NSURLSession!, dataTask: NSURLSessionDataTask!, didReceiveData data: NSData!) {

if let delegate = self[dataTask] as? Request.DataTaskDelegate {

delegate.URLSession(session, dataTask: dataTask, didReceiveData: data)

}

dataTaskDidReceiveData?(session, dataTask, data)

}

func URLSession(session: NSURLSession!, dataTask: NSURLSessionDataTask!, willCacheResponse proposedResponse: NSCachedURLResponse!, completionHandler: ((NSCachedURLResponse!) -> Void)!) {

var cachedResponse = proposedResponse

if dataTaskWillCacheResponse != nil {

cachedResponse = dataTaskWillCacheResponse!(session, dataTask, proposedResponse)

}

completionHandler(cachedResponse)

}

// MARK: NSURLSessionDownloadDelegate

func URLSession(session: NSURLSession, downloadTask: NSURLSessionDownloadTask, didFinishDownloadingToURL location: NSURL) {

if let delegate = self[downloadTask] as? Request.DownloadTaskDelegate {

delegate.URLSession(session, downloadTask: downloadTask, didFinishDownloadingToURL: location)

}

downloadTaskDidFinishDownloadingToURL?(session, downloadTask, location)

}

func URLSession(session: NSURLSession, downloadTask: NSURLSessionDownloadTask, didWriteData bytesWritten: Int64, totalBytesWritten: Int64, totalBytesExpectedToWrite: Int64) {

if let delegate = self[downloadTask] as? Request.DownloadTaskDelegate {

delegate.URLSession(session, downloadTask: downloadTask, didWriteData: bytesWritten, totalBytesWritten: totalBytesWritten, totalBytesExpectedToWrite: totalBytesExpectedToWrite)

}

downloadTaskDidWriteData?(session, downloadTask, bytesWritten, totalBytesWritten, totalBytesExpectedToWrite)

}

func URLSession(session: NSURLSession, downloadTask: NSURLSessionDownloadTask, didResumeAtOffset fileOffset: Int64, expectedTotalBytes: Int64) {

if let delegate = self[downloadTask] as? Request.DownloadTaskDelegate {

delegate.URLSession(session, downloadTask: downloadTask, didResumeAtOffset: fileOffset, expectedTotalBytes: expectedTotalBytes)

}

downloadTaskDidResumeAtOffset?(session, downloadTask, fileOffset, expectedTotalBytes)

}

// MARK: NSObject

override func respondsToSelector(selector: Selector) -> Bool {

switch selector {

case "URLSession:didBecomeInvalidWithError:":

return (sessionDidBecomeInvalidWithError != nil)

case "URLSession:didReceiveChallenge:completionHandler:":

return (sessionDidReceiveChallenge != nil)

case "URLSessionDidFinishEventsForBackgroundURLSession:":

return (sessionDidFinishEventsForBackgroundURLSession != nil)

case "URLSession:task:willPerformHTTPRedirection:newRequest:completionHandler:":

return (taskWillPerformHTTPRedirection != nil)

case "URLSession:dataTask:didReceiveResponse:completionHandler:":

return (dataTaskDidReceiveResponse != nil)

case "URLSession:dataTask:willCacheResponse:completionHandler:":

return (dataTaskWillCacheResponse != nil)

default:

return self.dynamicType.instancesRespondToSelector(selector)

}

}

}

}

// MARK: -

/\*\*

Responsible for sending a request and receiving the response and associated data from the server, as well as managing its underlying `NSURLSessionTask`.

\*/

public class Request {

private let delegate: TaskDelegate

/// The underlying task.

public var task: NSURLSessionTask { return delegate.task }

/// The session belonging to the underlying task.

public let session: NSURLSession

/// The request sent or to be sent to the server.

public var request: NSURLRequest { return task.originalRequest }

/// The response received from the server, if any.

public var response: NSHTTPURLResponse? { return task.response as? NSHTTPURLResponse }

/// The progress of the request lifecycle.

public var progress: NSProgress? { return delegate.progress }

private init(session: NSURLSession, task: NSURLSessionTask) {

self.session = session

switch task {

case is NSURLSessionUploadTask:

self.delegate = UploadTaskDelegate(task: task)

case is NSURLSessionDataTask:

self.delegate = DataTaskDelegate(task: task)

case is NSURLSessionDownloadTask:

self.delegate = DownloadTaskDelegate(task: task)

default:

self.delegate = TaskDelegate(task: task)

}

}

// MARK: Authentication

/\*\*

Associates an HTTP Basic credential with the request.

:param: user The user.

:param: password The password.

:returns: The request.

\*/

public func authenticate(#user: String, password: String) -> Self {

let credential = NSURLCredential(user: user, password: password, persistence: .ForSession)

return authenticate(usingCredential: credential)

}

/\*\*

Associates a specified credential with the request.

:param: credential The credential.

:returns: The request.

\*/

public func authenticate(usingCredential credential: NSURLCredential) -> Self {

delegate.credential = credential

return self

}

// MARK: Progress

/\*\*

Sets a closure to be called periodically during the lifecycle of the request as data is written to or read from the server.

- For uploads, the progress closure returns the bytes written, total bytes written, and total bytes expected to write.

- For downloads, the progress closure returns the bytes read, total bytes read, and total bytes expected to write.

:param: closure The code to be executed periodically during the lifecycle of the request.

:returns: The request.

\*/

public func progress(closure: ((Int64, Int64, Int64) -> Void)? = nil) -> Self {

if let uploadDelegate = delegate as? UploadTaskDelegate {

uploadDelegate.uploadProgress = closure

} else if let dataDelegate = delegate as? DataTaskDelegate {

dataDelegate.dataProgress = closure

} else if let downloadDelegate = delegate as? DownloadTaskDelegate {

downloadDelegate.downloadProgress = closure

}

return self

}

// MARK: Response

/\*\*

A closure used by response handlers that takes a request, response, and data and returns a serialized object and any error that occured in the process.

\*/

public typealias Serializer = (NSURLRequest, NSHTTPURLResponse?, NSData?) -> (AnyObject?, NSError?)

/\*\*

Creates a response serializer that returns the associated data as-is.

:returns: A data response serializer.

\*/

public class func responseDataSerializer() -> Serializer {

return { (request, response, data) in

return (data, nil)

}

}

/\*\*

Adds a handler to be called once the request has finished.

:param: completionHandler The code to be executed once the request has finished.

:returns: The request.

\*/

public func response(completionHandler: (NSURLRequest, NSHTTPURLResponse?, AnyObject?, NSError?) -> Void) -> Self {

return response(Request.responseDataSerializer(), completionHandler: completionHandler)

}

/\*\*

Adds a handler to be called once the request has finished.

:param: queue The queue on which the completion handler is dispatched.

:param: serializer The closure responsible for serializing the request, response, and data.

:param: completionHandler The code to be executed once the request has finished.

:returns: The request.

\*/

public func response(queue: dispatch\_queue\_t? = nil, serializer: Serializer, completionHandler: (NSURLRequest, NSHTTPURLResponse?, AnyObject?, NSError?) -> Void) -> Self {

dispatch\_async(delegate.queue) {

let (responseObject: AnyObject?, serializationError: NSError?) = serializer(self.request, self.response, self.delegate.data)

dispatch\_async(queue ?? dispatch\_get\_main\_queue()) {

completionHandler(self.request, self.response, responseObject, self.delegate.error ?? serializationError)

}

}

return self

}

/\*\*

Suspends the request.

\*/

public func suspend() {

task.suspend()

}

/\*\*

Resumes the request.

\*/

public func resume() {

task.resume()

}

/\*\*

Cancels the request.

\*/

public func cancel() {

if let downloadDelegate = delegate as? DownloadTaskDelegate {

downloadDelegate.downloadTask.cancelByProducingResumeData { (data) in

downloadDelegate.resumeData = data

}

} else {

task.cancel()

}

}

class TaskDelegate: NSObject, NSURLSessionTaskDelegate {

let task: NSURLSessionTask

let queue: dispatch\_queue\_t

let progress: NSProgress

var data: NSData? { return nil }

private(set) var error: NSError?

var credential: NSURLCredential?

var taskWillPerformHTTPRedirection: ((NSURLSession!, NSURLSessionTask!, NSHTTPURLResponse!, NSURLRequest!) -> (NSURLRequest!))?

var taskDidReceiveChallenge: ((NSURLSession!, NSURLSessionTask!, NSURLAuthenticationChallenge) -> (NSURLSessionAuthChallengeDisposition, NSURLCredential?))?

var taskDidSendBodyData: ((NSURLSession!, NSURLSessionTask!, Int64, Int64, Int64) -> Void)?

var taskNeedNewBodyStream: ((NSURLSession!, NSURLSessionTask!) -> (NSInputStream!))?

init(task: NSURLSessionTask) {

self.task = task

self.progress = NSProgress(totalUnitCount: 0)

self.queue = {

let label: String = "com.alamofire.task-\(task.taskIdentifier)"

let queue = dispatch\_queue\_create((label as NSString).UTF8String, DISPATCH\_QUEUE\_SERIAL)

dispatch\_suspend(queue)

return queue

}()

}

// MARK: NSURLSessionTaskDelegate

func URLSession(session: NSURLSession!, task: NSURLSessionTask!, willPerformHTTPRedirection response: NSHTTPURLResponse!, newRequest request: NSURLRequest!, completionHandler: ((NSURLRequest!) -> Void)!) {

var redirectRequest = request

if taskWillPerformHTTPRedirection != nil {

redirectRequest = taskWillPerformHTTPRedirection!(session, task, response, request)

}

completionHandler(redirectRequest)

}

func URLSession(session: NSURLSession!, task: NSURLSessionTask!, didReceiveChallenge challenge: NSURLAuthenticationChallenge!, completionHandler: ((NSURLSessionAuthChallengeDisposition, NSURLCredential!) -> Void)!) {

var disposition: NSURLSessionAuthChallengeDisposition = .PerformDefaultHandling

var credential: NSURLCredential?

if taskDidReceiveChallenge != nil {

(disposition, credential) = taskDidReceiveChallenge!(session, task, challenge)

} else {

if challenge.previousFailureCount > 0 {

disposition = .CancelAuthenticationChallenge

} else {

// TODO: Incorporate Trust Evaluation & TLS Chain Validation

switch challenge.protectionSpace.authenticationMethod! {

case NSURLAuthenticationMethodServerTrust:

credential = NSURLCredential(forTrust: challenge.protectionSpace.serverTrust)

default:

credential = self.credential ?? session.configuration.URLCredentialStorage?.defaultCredentialForProtectionSpace(challenge.protectionSpace)

}

if credential != nil {

disposition = .UseCredential

}

}

}

completionHandler(disposition, credential)

}

func URLSession(session: NSURLSession!, task: NSURLSessionTask!, needNewBodyStream completionHandler: ((NSInputStream!) -> Void)!) {

var bodyStream: NSInputStream?

if taskNeedNewBodyStream != nil {

bodyStream = taskNeedNewBodyStream!(session, task)

}

completionHandler(bodyStream)

}

func URLSession(session: NSURLSession!, task: NSURLSessionTask!, didCompleteWithError error: NSError!) {

if error != nil {

self.error = error

}

dispatch\_resume(queue)

}

}

class DataTaskDelegate: TaskDelegate, NSURLSessionDataDelegate {

var dataTask: NSURLSessionDataTask! { return task as NSURLSessionDataTask }

private var mutableData: NSMutableData

override var data: NSData? {

return mutableData

}

private var expectedContentLength: Int64?

var dataTaskDidReceiveResponse: ((NSURLSession!, NSURLSessionDataTask!, NSURLResponse!) -> (NSURLSessionResponseDisposition))?

var dataTaskDidBecomeDownloadTask: ((NSURLSession!, NSURLSessionDataTask!) -> Void)?

var dataTaskDidReceiveData: ((NSURLSession!, NSURLSessionDataTask!, NSData!) -> Void)?

var dataTaskWillCacheResponse: ((NSURLSession!, NSURLSessionDataTask!, NSCachedURLResponse!) -> (NSCachedURLResponse))?

var dataProgress: ((bytesReceived: Int64, totalBytesReceived: Int64, totalBytesExpectedToReceive: Int64) -> Void)?

override init(task: NSURLSessionTask) {

self.mutableData = NSMutableData()

super.init(task: task)

}

// MARK: NSURLSessionDataDelegate

func URLSession(session: NSURLSession!, dataTask: NSURLSessionDataTask!, didReceiveResponse response: NSURLResponse!, completionHandler: ((NSURLSessionResponseDisposition) -> Void)!) {

var disposition: NSURLSessionResponseDisposition = .Allow

expectedContentLength = response.expectedContentLength

if dataTaskDidReceiveResponse != nil {

disposition = dataTaskDidReceiveResponse!(session, dataTask, response)

}

completionHandler(disposition)

}

func URLSession(session: NSURLSession!, dataTask: NSURLSessionDataTask!, didBecomeDownloadTask downloadTask: NSURLSessionDownloadTask!) {

dataTaskDidBecomeDownloadTask?(session, dataTask)

}

func URLSession(session: NSURLSession!, dataTask: NSURLSessionDataTask!, didReceiveData data: NSData!) {

dataTaskDidReceiveData?(session, dataTask, data)

mutableData.appendData(data)

if let expectedContentLength = dataTask?.response?.expectedContentLength {

dataProgress?(bytesReceived: Int64(data.length), totalBytesReceived: Int64(mutableData.length), totalBytesExpectedToReceive: expectedContentLength)

}

}

func URLSession(session: NSURLSession!, dataTask: NSURLSessionDataTask!, willCacheResponse proposedResponse: NSCachedURLResponse!, completionHandler: ((NSCachedURLResponse!) -> Void)!) {

var cachedResponse = proposedResponse

if dataTaskWillCacheResponse != nil {

cachedResponse = dataTaskWillCacheResponse!(session, dataTask, proposedResponse)

}

completionHandler(cachedResponse)

}

}

}

// MARK: - Validation

extension Request {

/\*\*

A closure used to validate a request that takes a URL request and URL response, and returns whether the request was valid.

\*/

public typealias Validation = (NSURLRequest, NSHTTPURLResponse) -> (Bool)

/\*\*

Validates the request, using the specified closure.

If validation fails, subsequent calls to response handlers will have an associated error.

:param: validation A closure to validate the request.

:returns: The request.

\*/

public func validate(validation: Validation) -> Self {

dispatch\_async(delegate.queue) {

if self.response != nil && self.delegate.error == nil {

if !validation(self.request, self.response!) {

self.delegate.error = NSError(domain: AlamofireErrorDomain, code: -1, userInfo: nil)

}

}

}

return self

}

// MARK: Status Code

private class func response(response: NSHTTPURLResponse, hasAcceptableStatusCode statusCodes: [Int]) -> Bool {

return contains(statusCodes, response.statusCode)

}

/\*\*

Validates that the response has a status code in the specified range.

If validation fails, subsequent calls to response handlers will have an associated error.

:param: range The range of acceptable status codes.

:returns: The request.

\*/

public func validate(statusCode range: Range<Int>) -> Self {

return validate { (\_, response) in

return Request.response(response, hasAcceptableStatusCode: range.map({$0}))

}

}

/\*\*

Validates that the response has a status code in the specified array.

If validation fails, subsequent calls to response handlers will have an associated error.

:param: array The acceptable status codes.

:returns: The request.

\*/

public func validate(statusCode array: [Int]) -> Self {

return validate { (\_, response) in

return Request.response(response, hasAcceptableStatusCode: array)

}

}

// MARK: Content-Type

private struct MIMEType {

let type: String

let subtype: String

init(\_ string: String) {

let components = string.stringByTrimmingCharactersInSet(NSCharacterSet.whitespaceAndNewlineCharacterSet()).substringToIndex(string.rangeOfString(";")?.endIndex ?? string.endIndex).componentsSeparatedByString("/")

self.type = components.first!

self.subtype = components.last!

}

func matches(MIME: MIMEType) -> Bool {

switch (type, subtype) {

case ("\*", "\*"), ("\*", MIME.subtype), (MIME.type, "\*"), (MIME.type, MIME.subtype):

return true

default:

return false

}

}

}

private class func response(response: NSHTTPURLResponse, hasAcceptableContentType contentTypes: [String]) -> Bool {

if response.MIMEType != nil {

let responseMIMEType = MIMEType(response.MIMEType!)

for acceptableMIMEType in contentTypes.map({MIMEType($0)}) {

if acceptableMIMEType.matches(responseMIMEType) {

return true

}

}

}

return false

}

/\*\*

Validates that the response has a content type in the specified array.

If validation fails, subsequent calls to response handlers will have an associated error.

:param: contentType The acceptable content types, which may specify wildcard types and/or subtypes.

:returns: The request.

\*/

public func validate(contentType array: [String]) -> Self {

return validate {(\_, response) in

return Request.response(response, hasAcceptableContentType: array)

}

}

// MARK: Automatic

/\*\*

Validates that the response has a status code in the default acceptable range of 200...299, and that the content type matches any specified in the Accept HTTP header field.

If validation fails, subsequent calls to response handlers will have an associated error.

:returns: The request.

\*/

public func validate() -> Self {

let acceptableStatusCodes: Range<Int> = 200..<300

let acceptableContentTypes: [String] = {

if let accept = self.request.valueForHTTPHeaderField("Accept") {

return accept.componentsSeparatedByString(",")

}

return ["\*/\*"]

}()

return validate(statusCode: acceptableStatusCodes).validate(contentType: acceptableContentTypes)

}

}

// MARK: - Upload

extension Manager {

private enum Uploadable {

case Data(NSURLRequest, NSData)

case File(NSURLRequest, NSURL)

case Stream(NSURLRequest, NSInputStream)

}

private func upload(uploadable: Uploadable) -> Request {

var uploadTask: NSURLSessionUploadTask!

var stream: NSInputStream?

switch uploadable {

case .Data(let request, let data):

uploadTask = session.uploadTaskWithRequest(request, fromData: data)

case .File(let request, let fileURL):

uploadTask = session.uploadTaskWithRequest(request, fromFile: fileURL)

case .Stream(let request, var stream):

uploadTask = session.uploadTaskWithStreamedRequest(request)

}

let request = Request(session: session, task: uploadTask)

if stream != nil {

request.delegate.taskNeedNewBodyStream = { \_, \_ in

return stream

}

}

delegate[request.delegate.task] = request.delegate

if startRequestsImmediately {

request.resume()

}

return request

}

// MARK: File

/\*\*

Creates a request for uploading a file to the specified URL request.

If `startRequestsImmediately` is `true`, the request will have `resume()` called before being returned.

:param: URLRequest The URL request

:param: file The file to upload

:returns: The created upload request.

\*/

public func upload(URLRequest: URLRequestConvertible, file: NSURL) -> Request {

return upload(.File(URLRequest.URLRequest, file))

}

// MARK: Data

/\*\*

Creates a request for uploading data to the specified URL request.

If `startRequestsImmediately` is `true`, the request will have `resume()` called before being returned.

:param: URLRequest The URL request

:param: data The data to upload

:returns: The created upload request.

\*/

public func upload(URLRequest: URLRequestConvertible, data: NSData) -> Request {

return upload(.Data(URLRequest.URLRequest, data))

}

// MARK: Stream

/\*\*

Creates a request for uploading a stream to the specified URL request.

If `startRequestsImmediately` is `true`, the request will have `resume()` called before being returned.

:param: URLRequest The URL request

:param: stream The stream to upload

:returns: The created upload request.

\*/

public func upload(URLRequest: URLRequestConvertible, stream: NSInputStream) -> Request {

return upload(.Stream(URLRequest.URLRequest, stream))

}

}

extension Request {

class UploadTaskDelegate: DataTaskDelegate {

var uploadTask: NSURLSessionUploadTask! { return task as NSURLSessionUploadTask }

var uploadProgress: ((Int64, Int64, Int64) -> Void)!

// MARK: NSURLSessionTaskDelegate

func URLSession(session: NSURLSession!, task: NSURLSessionTask!, didSendBodyData bytesSent: Int64, totalBytesSent: Int64, totalBytesExpectedToSend: Int64) {

if uploadProgress != nil {

uploadProgress(bytesSent, totalBytesSent, totalBytesExpectedToSend)

}

progress.totalUnitCount = totalBytesExpectedToSend

progress.completedUnitCount = totalBytesSent

}

}

}

// MARK: - Download

extension Manager {

private enum Downloadable {

case Request(NSURLRequest)

case ResumeData(NSData)

}

private func download(downloadable: Downloadable, destination: (NSURL, NSHTTPURLResponse) -> (NSURL)) -> Request {

var downloadTask: NSURLSessionDownloadTask!

switch downloadable {

case .Request(let request):

downloadTask = session.downloadTaskWithRequest(request)

case .ResumeData(let resumeData):

downloadTask = session.downloadTaskWithResumeData(resumeData)

}

let request = Request(session: session, task: downloadTask)

if let downloadDelegate = request.delegate as? Request.DownloadTaskDelegate {

downloadDelegate.downloadTaskDidFinishDownloadingToURL = { (session, downloadTask, URL) in

return destination(URL, downloadTask.response as NSHTTPURLResponse)

}

}

delegate[request.delegate.task] = request.delegate

if startRequestsImmediately {

request.resume()

}

return request

}

// MARK: Request

/\*\*

Creates a request for downloading from the specified URL request.

If `startRequestsImmediately` is `true`, the request will have `resume()` called before being returned.

:param: URLRequest The URL request

:param: destination The closure used to determine the destination of the downloaded file.

:returns: The created download request.

\*/

public func download(URLRequest: URLRequestConvertible, destination: (NSURL, NSHTTPURLResponse) -> (NSURL)) -> Request {

return download(.Request(URLRequest.URLRequest), destination: destination)

}

// MARK: Resume Data

/\*\*

Creates a request for downloading from the resume data produced from a previous request cancellation.

If `startRequestsImmediately` is `true`, the request will have `resume()` called before being returned.

:param: resumeData The resume data. This is an opaque data blob produced by `NSURLSessionDownloadTask` when a task is cancelled. See `NSURLSession -downloadTaskWithResumeData:` for additional information.

:param: destination The closure used to determine the destination of the downloaded file.

:returns: The created download request.

\*/

public func download(resumeData: NSData, destination: Request.DownloadFileDestination) -> Request {

return download(.ResumeData(resumeData), destination: destination)

}

}

extension Request {

/\*\*

A closure executed once a request has successfully completed in order to determine where to move the temporary file written to during the download process. The closure takes two arguments: the temporary file URL and the URL response, and returns a single argument: the file URL where the temporary file should be moved.

\*/

public typealias DownloadFileDestination = (NSURL, NSHTTPURLResponse) -> (NSURL)

/\*\*

Creates a download file destination closure which uses the default file manager to move the temporary file to a file URL in the first available directory with the specified search path directory and search path domain mask.

:param: directory The search path directory. `.DocumentDirectory` by default.

:param: domain The search path domain mask. `.UserDomainMask` by default.

:returns: A download file destination closure.

\*/

public class func suggestedDownloadDestination(directory: NSSearchPathDirectory = .DocumentDirectory, domain: NSSearchPathDomainMask = .UserDomainMask) -> DownloadFileDestination {

return { (temporaryURL, response) -> (NSURL) in

if let directoryURL = NSFileManager.defaultManager().URLsForDirectory(.DocumentDirectory, inDomains: .UserDomainMask)[0] as? NSURL {

return directoryURL.URLByAppendingPathComponent(response.suggestedFilename!)

}

return temporaryURL

}

}

class DownloadTaskDelegate: TaskDelegate, NSURLSessionDownloadDelegate {

var downloadTask: NSURLSessionDownloadTask! { return task as NSURLSessionDownloadTask }

var downloadProgress: ((Int64, Int64, Int64) -> Void)?

var resumeData: NSData?

override var data: NSData? { return resumeData }

var downloadTaskDidFinishDownloadingToURL: ((NSURLSession!, NSURLSessionDownloadTask!, NSURL) -> (NSURL))?

var downloadTaskDidWriteData: ((NSURLSession!, NSURLSessionDownloadTask!, Int64, Int64, Int64) -> Void)?

var downloadTaskDidResumeAtOffset: ((NSURLSession!, NSURLSessionDownloadTask!, Int64, Int64) -> Void)?

// MARK: NSURLSessionDownloadDelegate

func URLSession(session: NSURLSession, downloadTask: NSURLSessionDownloadTask, didFinishDownloadingToURL location: NSURL) {

if downloadTaskDidFinishDownloadingToURL != nil {

let destination = downloadTaskDidFinishDownloadingToURL!(session, downloadTask, location)

var fileManagerError: NSError?

NSFileManager.defaultManager().moveItemAtURL(location, toURL: destination, error: &fileManagerError)

if fileManagerError != nil {

error = fileManagerError

}

}

}

func URLSession(session: NSURLSession!, downloadTask: NSURLSessionDownloadTask!, didWriteData bytesWritten: Int64, totalBytesWritten: Int64, totalBytesExpectedToWrite: Int64) {

downloadTaskDidWriteData?(session, downloadTask, bytesWritten, totalBytesWritten, totalBytesExpectedToWrite)

downloadProgress?(bytesWritten, totalBytesWritten, totalBytesExpectedToWrite)

progress.totalUnitCount = totalBytesExpectedToWrite

progress.completedUnitCount = totalBytesWritten

}

func URLSession(session: NSURLSession!, downloadTask: NSURLSessionDownloadTask!, didResumeAtOffset fileOffset: Int64, expectedTotalBytes: Int64) {

downloadTaskDidResumeAtOffset?(session, downloadTask, fileOffset, expectedTotalBytes)

progress.totalUnitCount = expectedTotalBytes

progress.completedUnitCount = fileOffset

}

}

}

// MARK: - Printable

extension Request: Printable {

/// The textual representation used when written to an `OutputStreamType`, which includes the HTTP method and URL, as well as the response status code if a response has been received.

public var description: String {

var components: [String] = []

if request.HTTPMethod != nil {

components.append(request.HTTPMethod!)

}

components.append(request.URL.absoluteString!)

if response != nil {

components.append("(\(response!.statusCode))")

}

return join(" ", components)

}

}

extension Request: DebugPrintable {

func cURLRepresentation() -> String {

var components: [String] = ["$ curl -i"]

let URL = request.URL

if request.HTTPMethod != nil && request.HTTPMethod != "GET" {

components.append("-X \(request.HTTPMethod!)")

}

if let credentialStorage = self.session.configuration.URLCredentialStorage {

let protectionSpace = NSURLProtectionSpace(host: URL.host!, port: URL.port?.integerValue ?? 0, `protocol`: URL.scheme!, realm: URL.host!, authenticationMethod: NSURLAuthenticationMethodHTTPBasic)

if let credentials = credentialStorage.credentialsForProtectionSpace(protectionSpace)?.values.array {

for credential: NSURLCredential in (credentials as [NSURLCredential]) {

components.append("-u \(credential.user!):\(credential.password!)")

}

} else {

if let credential = delegate.credential {

components.append("-u \(credential.user!):\(credential.password!)")

}

}

}

if let cookieStorage = session.configuration.HTTPCookieStorage {

if let cookies = cookieStorage.cookiesForURL(URL) as? [NSHTTPCookie] {

if !cookies.isEmpty {

let string = cookies.reduce(""){ $0 + "\($1.name)=\($1.value);" }

components.append("-b \"\(string.substringToIndex(string.endIndex.predecessor()))\"")

}

}

}

for (field, value) in request.allHTTPHeaderFields! {

switch field {

case "Cookie":

continue

default:

components.append("-H \"\(field): \(value)\"")

}

}

for (field, value) in session.configuration.HTTPAdditionalHeaders! {

switch field {

case "Cookie":

continue

default:

components.append("-H \"\(field): \(value)\"")

}

}

if let HTTPBody = request.HTTPBody {

if let escapedBody = NSString(data: HTTPBody, encoding: NSUTF8StringEncoding)?.stringByReplacingOccurrencesOfString("\"", withString: "\\\"") {

components.append("-d \"\(escapedBody)\"")

}

}

components.append("\"\(URL.absoluteString!)\"")

return join(" \\\n\t", components)

}

/// The textual representation used when written to an `OutputStreamType`, in the form of a cURL command.

public var debugDescription: String {

return cURLRepresentation()

}

}

// MARK: - Response Serializers

// MARK: String

extension Request {

/\*\*

Creates a response serializer that returns a string initialized from the response data with the specified string encoding.

:param: encoding The string encoding. `NSUTF8StringEncoding` by default.

:returns: A string response serializer.

\*/

public class func stringResponseSerializer(encoding: NSStringEncoding = NSUTF8StringEncoding) -> Serializer {

return { (\_, \_, data) in

let string = NSString(data: data!, encoding: encoding)

return (string, nil)

}

}

/\*\*

Adds a handler to be called once the request has finished.

:param: completionHandler A closure to be executed once the request has finished. The closure takes 4 arguments: the URL request, the URL response, if one was received, the string, if one could be created from the URL response and data, and any error produced while creating the string.

:returns: The request.

\*/

public func responseString(completionHandler: (NSURLRequest, NSHTTPURLResponse?, String?, NSError?) -> Void) -> Self {

return responseString(completionHandler: completionHandler)

}

/\*\*

Adds a handler to be called once the request has finished.

:param: encoding The string encoding. `NSUTF8StringEncoding` by default.

:param: completionHandler A closure to be executed once the request has finished. The closure takes 4 arguments: the URL request, the URL response, if one was received, the string, if one could be created from the URL response and data, and any error produced while creating the string.

:returns: The request.

\*/

public func responseString(encoding: NSStringEncoding = NSUTF8StringEncoding, completionHandler: (NSURLRequest, NSHTTPURLResponse?, String?, NSError?) -> Void) -> Self {

return response(serializer: Request.stringResponseSerializer(encoding: encoding), completionHandler: { request, response, string, error in

completionHandler(request, response, string as? String, error)

})

}

}

// MARK: JSON

extension Request {

/\*\*

Creates a response serializer that returns a JSON object constructed from the response data using `NSJSONSerialization` with the specified reading options.

:param: options The JSON serialization reading options. `.AllowFragments` by default.

:returns: A JSON object response serializer.

\*/

public class func JSONResponseSerializer(options: NSJSONReadingOptions = .AllowFragments) -> Serializer {

return { (request, response, data) in

if data == nil {

return (nil, nil)

}

var serializationError: NSError?

let JSON: AnyObject? = NSJSONSerialization.JSONObjectWithData(data!, options: options, error: &serializationError)

return (JSON, serializationError)

}

}

/\*\*

Adds a handler to be called once the request has finished.

:param: completionHandler A closure to be executed once the request has finished. The closure takes 4 arguments: the URL request, the URL response, if one was received, the JSON object, if one could be created from the URL response and data, and any error produced while creating the JSON object.

:returns: The request.

\*/

public func responseJSON(completionHandler: (NSURLRequest, NSHTTPURLResponse?, AnyObject?, NSError?) -> Void) -> Self {

return responseJSON(completionHandler: completionHandler)

}

/\*\*

Adds a handler to be called once the request has finished.

:param: options The JSON serialization reading options. `.AllowFragments` by default.

:param: completionHandler A closure to be executed once the request has finished. The closure takes 4 arguments: the URL request, the URL response, if one was received, the JSON object, if one could be created from the URL response and data, and any error produced while creating the JSON object.

:returns: The request.

\*/

public func responseJSON(options: NSJSONReadingOptions = .AllowFragments, completionHandler: (NSURLRequest, NSHTTPURLResponse?, AnyObject?, NSError?) -> Void) -> Self {

return response(serializer: Request.JSONResponseSerializer(options: options), completionHandler: { (request, response, JSON, error) in

completionHandler(request, response, JSON, error)

})

}

}

// MARK: Property List

extension Request {

/\*\*

Creates a response serializer that returns an object constructed from the response data using `NSPropertyListSerialization` with the specified reading options.

:param: options The property list reading options. `0` by default.

:returns: A property list object response serializer.

\*/

public class func propertyListResponseSerializer(options: NSPropertyListReadOptions = 0) -> Serializer {

return { (request, response, data) in

if data == nil {

return (nil, nil)

}

var propertyListSerializationError: NSError?

let plist: AnyObject? = NSPropertyListSerialization.propertyListWithData(data!, options: options, format: nil, error: &propertyListSerializationError)

return (plist, propertyListSerializationError)

}

}

/\*\*

Adds a handler to be called once the request has finished.

:param: completionHandler A closure to be executed once the request has finished. The closure takes 4 arguments: the URL request, the URL response, if one was received, the property list, if one could be created from the URL response and data, and any error produced while creating the property list.

:returns: The request.

\*/

public func responsePropertyList(completionHandler: (NSURLRequest, NSHTTPURLResponse?, AnyObject?, NSError?) -> Void) -> Self {

return responsePropertyList(completionHandler: completionHandler)

}

/\*\*

Adds a handler to be called once the request has finished.

:param: options The property list reading options. `0` by default.

:param: completionHandler A closure to be executed once the request has finished. The closure takes 4 arguments: the URL request, the URL response, if one was received, the property list, if one could be created from the URL response and data, and any error produced while creating the property list.

:returns: The request.

\*/

public func responsePropertyList(options: NSPropertyListReadOptions = 0, completionHandler: (NSURLRequest, NSHTTPURLResponse?, AnyObject?, NSError?) -> Void) -> Self {

return response(serializer: Request.propertyListResponseSerializer(options: options), completionHandler: { (request, response, plist, error) in

completionHandler(request, response, plist, error)

})

}

}

// MARK: - Convenience -

private func URLRequest(method: Method, URL: URLStringConvertible) -> NSURLRequest {

let mutableURLRequest = NSMutableURLRequest(URL: NSURL(string: URL.URLString)!)

mutableURLRequest.HTTPMethod = method.rawValue

return mutableURLRequest

}

// MARK: - Request

/\*\*

Creates a request using the shared manager instance for the specified method, URL string, parameters, and parameter encoding.

:param: method The HTTP method.

:param: URLString The URL string.

:param: parameters The parameters. `nil` by default.

:param: encoding The parameter encoding. `.URL` by default.

:returns: The created request.

\*/

public func request(method: Method, URLString: URLStringConvertible, parameters: [String: AnyObject]? = nil, encoding: ParameterEncoding = .URL) -> Request {

return request(encoding.encode(URLRequest(method, URLString), parameters: parameters).0)

}

/\*\*

Creates a request using the shared manager instance for the specified URL request.

If `startRequestsImmediately` is `true`, the request will have `resume()` called before being returned.

:param: URLRequest The URL request

:returns: The created request.

\*/

public func request(URLRequest: URLRequestConvertible) -> Request {

return Manager.sharedInstance.request(URLRequest.URLRequest)

}

// MARK: - Upload

// MARK: File

/\*\*

Creates an upload request using the shared manager instance for the specified method, URL string, and file.

:param: method The HTTP method.

:param: URLString The URL string.

:param: file The file to upload.

:returns: The created upload request.

\*/

public func upload(method: Method, URLString: URLStringConvertible, file: NSURL) -> Request {

return Manager.sharedInstance.upload(URLRequest(method, URLString), file: file)

}

/\*\*

Creates an upload request using the shared manager instance for the specified URL request and file.

:param: URLRequest The URL request.

:param: file The file to upload.

:returns: The created upload request.

\*/

public func upload(URLRequest: URLRequestConvertible, file: NSURL) -> Request {

return Manager.sharedInstance.upload(URLRequest, file: file)

}

// MARK: Data

/\*\*

Creates an upload request using the shared manager instance for the specified method, URL string, and data.

:param: method The HTTP method.

:param: URLString The URL string.

:param: data The data to upload.

:returns: The created upload request.

\*/

public func upload(method: Method, URLString: URLStringConvertible, data: NSData) -> Request {

return Manager.sharedInstance.upload(URLRequest(method, URLString), data: data)

}

/\*\*

Creates an upload request using the shared manager instance for the specified URL request and data.

:param: URLRequest The URL request.

:param: data The data to upload.

:returns: The created upload request.

\*/

public func upload(URLRequest: URLRequestConvertible, data: NSData) -> Request {

return Manager.sharedInstance.upload(URLRequest, data: data)

}

// MARK: Stream

/\*\*

Creates an upload request using the shared manager instance for the specified method, URL string, and stream.

:param: method The HTTP method.

:param: URLString The URL string.

:param: stream The stream to upload.

:returns: The created upload request.

\*/

public func upload(method: Method, URLString: URLStringConvertible, stream: NSInputStream) -> Request {

return Manager.sharedInstance.upload(URLRequest(method, URLString), stream: stream)

}

/\*\*

Creates an upload request using the shared manager instance for the specified URL request and stream.

:param: URLRequest The URL request.

:param: stream The stream to upload.

:returns: The created upload request.

\*/

public func upload(URLRequest: URLRequestConvertible, stream: NSInputStream) -> Request {

return Manager.sharedInstance.upload(URLRequest, stream: stream)

}

// MARK: - Download

// MARK: URL Request

/\*\*

Creates a download request using the shared manager instance for the specified method and URL string.

:param: method The HTTP method.

:param: URLString The URL string.

:param: destination The closure used to determine the destination of the downloaded file.

:returns: The created download request.

\*/

public func download(method: Method, URLString: URLStringConvertible, destination: Request.DownloadFileDestination) -> Request {

return Manager.sharedInstance.download(URLRequest(method, URLString), destination: destination)

}

/\*\*

Creates a download request using the shared manager instance for the specified URL request.

:param: URLRequest The URL request.

:param: destination The closure used to determine the destination of the downloaded file.

:returns: The created download request.

\*/

public func download(URLRequest: URLRequestConvertible, destination: Request.DownloadFileDestination) -> Request {

return Manager.sharedInstance.download(URLRequest, destination: destination)

}

// MARK: Resume Data

/\*\*

Creates a request using the shared manager instance for downloading from the resume data produced from a previous request cancellation.

:param: resumeData The resume data. This is an opaque data blob produced by `NSURLSessionDownloadTask` when a task is cancelled. See `NSURLSession -downloadTaskWithResumeData:` for additional information.

:param: destination The closure used to determine the destination of the downloaded file.

:returns: The created download request.

\*/

public func download(resumeData data: NSData, destination: Request.DownloadFileDestination) -> Request {

return Manager.sharedInstance.download(data, destination: destination)

}