Communicating with HTTP Servers

Moritz Haarmann, *Software for mobile devices*

1 Communicating with HTTP Servers

In this guide we'll explore some ways to retrieve and send information from a HTTP server. We'll only work with built-in frameworks here to get you started.

Some of the techniques discussed here are mostly here for the sake of completeness, yet I think it's really important to provide a thorough overview to understand the possibilities and limitations of the libraries.

1.1 Synchronous vs. Asynchronous

If you're familiar with the terms and know that you shouldn't ever pollute the main thread with IO-related tasks, you can just skip this section. If not it's really important to understand those concepts to build responsive, good apps.

All UI-related tasks are performed on the so-called main thread of an iOS app. This thread is executed with a high priority to make interaction fluid, fast and non-stuttering. The main thread should never be blocked to read a file or perform

long-running computations – those tasks really must be put on a background thread using some of the means provided. When it comes to networking, which really is one of the slowest IO-operations we usually handle, staying on the main thread is generally the worst idea one can have.

We'll talk about operation queues and Grand Central Dispatch, a framework for performing work in the background, later in this book, but for now just remember that synchronous means that a task will be performed on the calling thread, while asynchronous means it performs it's work somewhere on another thread (where exactly doesn't matter that much) and therefore doesn't freeze the UI – which is what happens with blocking or synchronous calls.

1.2 5 Ways to fetch a HTTP resource using only system tools.

1.2.1 NSString stringWithContentsOfURL:encoding:error

The simplest way to just fetch a resource as a NSString-object is by using the class method stringWithContentsOfURL:encoding:error on NSString, like

```
NSError *error;

NSURL *url = [NSURL URLWithString:kServerURL];

NSString *response = [NSString stringWithContentsOfURL:url encoding:NS

if (response){
    NSLog(@"%@", response);
} else if (error){
    NSLog(@"%@", [error localizedDescription]);
}
```

Works, and if it doesn't, we have some error information. This call is synchronous, meaning it will block the runloop that called the method until it is finished loading the data.

1.2.2 NSData dataWithContentsOfURL

String content is a bit narrow minded, in case of images or other binary data NSString is far from the right class to represent this kind of data. The example looks almost like the one above:

```
NSError *error;
```

```
NSURL *url = [NSURL URLWithString:kServerURL];

NSData *response = [NSData dataWithContentsOfURL:url options:0 error:&e

if (response){
    NSLog(@"%@", [[NSString alloc] initWithData:response encoding:NSUT
} else if (error){
    NSLog(@"%@", [error localizedDescription]);
}
```

As the previous example, this call is performed synchronous.

1.2.3 NSURLConnection, synchronous call

iOS comes with a set of classes to handle URL Loading. One of those is NSURLConnection. NSURLConnection allows for both synchronous and asynchronous URL loading, in this case we'll have a look at synchronous loading.

```
NSError *error = nil;

NSURL *url = [NSURL URLWithString:kServerURL];

NSURLRequest *request = [NSURLRequest requestWithURL:url];

NSURLResponse *response = nil;

NSData *data = [NSURLConnection sendSynchronousRequest:request returnin

if (response != nil){
    NSLog(@"Received a response, length: %d", data.length);
} else {
    if (error!=nil){
        NSLog(@"%@", [error localizedDescription]);
    }
}
```

As you can see, the amount of code required is considerably higher than in the previous examples. What is it then that makes people use this construct? The reason is: control. While the two former examples illustrated an easy-to-use way of just grabbing data without caring too much about what happens internally,

this approach allows us to customize the request to exactly fit our needs and expectations, as we'll see in the next example.

1.2.4 NSURLConnection with asynchronous loading and a custom header

In this example, we'll retrieve a resource with an additional request header set. In this example, the connection is also asynchronous and thus not blocking the main thread.

```
-(void)loadContents {
    NSURL *url = [NSURL URLWithString:kServerURL];

    NSMutableURLRequest *request = [NSMutableURLRequest requestWithURL:
    [request addValue:@"1" forHTTPHeaderField:@"X-App-Debug"];

    self.responseData = [NSMutableData new];

    NSURLConnection *connection = [NSURLConnection connectionWithReques

    [connection start];
}

-(void)connection:(NSURLConnection *)connection didReceiveData:(NSData
    [self.responseData appendData:data];
}

-(void)connection:(NSURLConnection *)connection didReceiveResponse:(NS
    self.response = response;
}

-(void)connection:(NSURLConnection *)connection didFailWithError:(NSEr
    NSLog(@"Connection Failed: %@", error);
}

-(void)connectionDidFinishLoading:(NSURLConnection *)connection {
    NSLog(@"Connection finished, received %d bytes of data.", self.resp
}
```

This is by far the longest example here, since we have to do a lot of stuff here for ourselves. First, we assume that there are to instance variables, one named response

to hold an NSURLResponse object, and the other named responseDatastoring a NSMutableData object that will store the received body data.

After starting the connection, it will call the delegate methods on each event and in the end either make a call to the connection: didFailWithError : or connectionDidFinishLoading: delegate method defined in the NSURLConnectionDataDelegate-protocol.

1.2.5 NSURLSession

iOS7 has changed a lot, also in terms of what is possible when it comes to downloading data. To leverage a whole set of new features like background downloads, Apple included a new class NSURLSession starting with iOS7. It's obvious from the naming that a NSURLSession represents a whole session, not only a single request. This is a practical approach, since most cases require multiple requests to be made, all configured in a similar way.

The first supported type of session is a default session that closely resembles the behavior of the good ol' NSURLConnection we've seen above. Since NSURLConnections, and inspired by that, also default sessions use a system wide URL cache to cache responses as discussed in the first chapter, there is an opposite type of session, called *ephemeral session* that doesn't cache it's contents. This might be useful in some situations where caching is not desired.

The third type of session is the most interesting one, called download session. It's purpose is to download files to disk. This is nothing especially fancy, but it downloads those files also while an app is suspended or in background.

For this example, we'll use a default session configuration to achieve the same result as in the previous samples.

To use a NSURLSession, we have to do some coding: a working example consists of an NSURLSessionConfiguration-object, configuring the Session, the resulting NSURLSession and NSURLSessionTasks that encapsulate the single requests.

```
NSURL *url = [NSURL URLWithString:kServerURL];

NSURLRequest *request = [NSURLRequest requestWithURL:url];

NSURLSessionConfiguration *sessionConfiguration = [NSURLSessionConfigu

sessionConfiguration.HTTPAdditionalHeaders = @{@"X-App-Debug": @"1"};

NSURLSession *session = [NSURLSession sessionWithConfiguration:sessionC

NSURLSessionTask *task = [session dataTaskWithRequest:request completio
        if (response != nil){
```

```
            NSLog(@"Received a response, length: %d", data.length);
        } else {
            if (error!=nil){
                NSLog(@"%@", [error localizedDescription]);
            }
        }
}];

[task resume];
```

The last line, [task resume] might seem odd, but the created task is not yet running, calling resume will fix that.

As you've certainly noticed, we are setting the additional request header be assigning a NSDictionary to the HTTPAdditionalHeader property of the session configuration. That way, all requests performed with this configuration will have that header set. But not only headers can be customized, basically any aspect of the client-server communication and behavior can be adapted to the requirements:

- Network types used (Cellular, WiFi)
- Cookie Policies
- Caching Policies and the URL Cache to use
- Maximum number of connections per host
- Timeouts for both the requests and responses
- Proxies
- Whether pipelining[1] should be used.
- Authentication challenges
- Supported TLS Protocols

Not much left to desire. If you are starting to develop a new app or migrating an existing app to be iOS7 only, I strongly suggest you stick to NSURLSession for performing HTTP communication, as it comes with almost anything you'll ever need.

In the remainder of this book, we'll use NSURLSession for communicating with servers, as the other means are either outdated (NSURLConnection) or not customizable enough for use in serious app development.

[1] Pipelining is a HTTP/1.1 technique that breaks the request-response-request-response pattern by allowing 2 or more requests to be performed on the same connection without waiting for each response before the next request is made. This allows, sometimes, for a quicker loading of the combined resources.

1.3 Authentication

Most applications won't work without any form of authorization: Twitter, Facebook, ToDo-lists require you to log in to see your data. This log-in usually consists of a username and a password.

Since providing every client app that uses a service requiring authorizating with the password is considered a bad practice, techniques to overcome this problem have evolved and became widespread. OAuth, for *open standard for authorization* is one of those techniques with a proven track record and excellent client libraries to simplify its integration into existing projects. We'll discuss how to perform user/password based authentication as well later, but you are strongly advised against using it in production scenarios.

1.3.1 OAuth

OAuth can be complicated to grasp at first. I'll try to explain it as simple as possible without leaving anything important out.

The idea behind OAuth is that it would be desirable to grant access to certain protected resources on behalf of the user. To avoid sharing the users password, und thus giving up control over what applications use protected data and the ability to revoke control in the future, the applications requiring access to those protected resources are given a set of credentials that only they can use to access those resources.

The authentication flow consists, simplified, of three steps. In the first step the client, our app, creates a so-called request token. Don't get fooled here, this has nothing to do with HTTP request headers. Using this token (and a pre-generated application token uniquely identifying our application to the third-party server we'd like to access) our app opens a URL on the third-party-server displaying an authentication dialog. The user enters her username and password only there, on the property of the third-party, and authorizes our app to use her protected resources. Once she does so, the third-party app redirects (using URL-Schemes) the user back into our app, where we can now exchange the generated request-token into an app token we can use to authenticate requests. Those authenticated requests are created using both our app-id and a pre-generated app secret that is known only to us and the third-party provider. Only the combination of app id, app token and app secret allows access to the protected resources.

The users profit from this workflow in many ways. Contrary to the just-save-my-password approach, OAuth let's users revoke access of certain apps to their protected resources (maybe because they have been compromised or don't meet the users expectations.). Additionally, by only being required to entering the password

once, on the trusted third-party page, the amount of trust required to use the product is not as high compared to a simple password-only approach.

The question is: how to implement OAuth authentication in your iOS-App? I originally planned to start with a hand-crafted example of how to use OAuth without relying directly on any library. After writing the code, I dropped the idea (Only the part signing a request has, without comments, 80 lines.). Relying on libraries is crucial when writing serious code, and OAuth is definitely one of the spots where using third-party tools is basically the only sane way.

This is a volatile field, especially when it comes to third-party libraries. As a rule of thumb: if it works for you, the library is fine.

1.4 *Simple* Twitter authentication using OAuth

In this example, we'll explore how to implement a typical OAuth 1.0 flow using Twitter as our service provider. We'll, ignoring all principles of software design, use just one view controller and our goal is to get a dump of the user timeline once we are authorized. We won't save any credentials, not display anything besides the login form and be ignorant of memory management.

To start, create a new iPhone Project from the single view template. Name it whatever you want. once it's saved, we need to create a Podfile referencing OAuthCore, a library by *atebits*. This library implements some of the functionality required to sign a request to make it work for OAuth. Since this part is especially tedious and painful, I decided to leave it out as an exercise to the super curious reader to implement it herself.

The contents of the Podfile are simple:

```
platform :ios, '7.0'

pod "OAuthCore"
```

Once you've created the Podfile in your project's root folder, fire up a terminal and run pod install in the same folder. If you don't have CocoaPods installed at this point: their website has a pretty solid guide that will get you up to speed in pretty much no time.

Do as advised and use the newly created .xcworkspace from now on. Open it, we have some coding to do.

What we are going to do is to place a UIWebView in our precreated View Controller, which, depending on your class prefix you choose, is named something like XYViewController. We'll then connect the web view to an outlet in our XYViewController. We also need to set the web view's delegate outlet to our view controller to be able to intercept redirects later in the process.

> You are invited to just clone and run the completed example. But it may be *oddly satisfying* to just try it yourself.

For this demo to be work on your machine, you need to create a twitter client application at dev.twitter.com. To do so, you need to be a registered users. Once you have finished creating your application there, you can extract a bunch of important values from the dashboard there. We will use those values to #define some constants we'll use a lot[2]. Just put those value on the top of your XYViewController.m

```
    #define kConsumerKey @"TdhoA5JeKMtLWmmsygl4BA"
#define kConsumerSecret @"9uKsjwRI7Hp9zwC5sDWtFuEXdJl2TmfIHmHZjeaXJRs"
#define kTwitterRequestTokenURLString @"https://api.twitter.com/oauth/r
#define kTwitterAuthorizationURLString @"https://api.twitter.com/oauth/
#define kTwitterExchangeTokenURLString @"https://api.twitter.com/oauth/
```

It's also a good idea to have one NSURLSession ready-to-use for our networking tasks. We'll create one using the default configuration in the viewDidLoad-method. Make sure to have a proper instance variable in place to hold the session. We are using the default configuration since it will be completely sufficient for what we are doing.

```
        - (void)viewDidLoad
{
    [super viewDidLoad];

    self.session = [NSURLSession sessionWithConfiguration:[NSURLSessio

            self.webView.delegate = self;
}
```

Right now, we can start with the real OAuth flow. The first step is to obtain a so-called request token from the server. This token is then used during the login procedure and later exchanged for a real, so-called access token. Let's obtain the request token. I've created a method called obtainRequestToken that contains the functionality. As you'll see, we have quite some work to do.

```
-(void)obtainRequestToken {
NSMutableURLRequest *request = [NSMutableURLRequest requestWithURL:
request.HTTPMethod = @"POST";

NSString *authorizationHeaderValue = OAuthorizationHeader(request.U
```

[2] You can, if you want to, of course use NSString *const @"anything" for defining those values. Using #define here is just a matter of readability.

```
    [request addValue:authorizationHeaderValue forHTTPHeaderField:@"Au

    NSURLSessionDataTask *requestTokenTask = [self.session dataTaskWith
        if (data.length){
            NSString *payload = [[NSString alloc] initWithData:data enc
            NSDictionary *contentDictionary = [NSURL ab_parseURLQueryS

            self.token = contentDictionary[@"oauth_token"];
            self.tokenSecret = contentDictionary[@"oauth_token_secret"]

            [self loadLoginFormInWebview];
        } else {
            // We would handle this error here.
        }
    }];

    [requestTokenTask resume];
}
```

Let's go over the important parts here. The first thing we do is to create a mutable URL request using the request token url Twitter provided us. We set the HTTP method to POST. Then we pass almost all request information into a method called OAuthorizationHeader, which is one of the methods provided by OAuthCore. This method creates a header value that is required by OAuth, using all the information provided: the URL, parameters and of course the consumer secret and key of our app.

We add this authorization header to the request and then start the actual network operation using an instance of NSURLSessionDataTask.

Using the completion handler we check to see if the data is actually not empty, as we'll try to extract the payload from the response content. In case of success, this payload contains the request token as well as the request secret which is used to sign the requests we make until we do have the access token. It's important to remember that only you as an developer and Twitter have access to the secret component at any time, making it possible to use this to sign a request and thereby proving that a legitimate instance (you) created it in the first place. We need to save the token and token secret to instance variables to have them ready when we need them. Both of them are strings.

The method we are using to extract a dictionary of key-value pairs from the response content is also taken from OAuthCore, ab_parseURLQueryString:payload.

It's curiously implemented as a category on NSURL. If successful, it returns an instance of NSDictionary and we can extract the token and the secret as shown above. We can then show the login form, using our request token, in the UIWebView we've created. This is all done in the method loadLoginForm.

```
-(void)loadLoginForm {
NSString *loginURLString = [kTwitterAuthorizationURLString stringBy

[self.webView loadRequest:[NSURLRequest requestWithURL:[NSURL URLW
}
```

By simply appending a query part to the authorization URL given by twitter containing the request token we just received. We then pass this to the web view, which will load and display the resulting page. If you've done everything right until now, a login form showing also your applications chosen name when you created it should appear.

If you enter your credentials at this point, nothing happens. Like, really nothing. OAuth works, at this point, by redirecting the user after a successful login to a predefined URL. You entered that URL when you registered your app with Twitter. So, what we do now is to implement one of UIWebView's delegate methods, webView:shouldStartLoadRequest:navigationType, that allows us to intercept and, if desired, cancel the loading of certain requests. We'll just intercept the request that is the redirect by matching everything that the webview loads against our predefined redirect url, like so:

```
-(BOOL)webView:(UIWebView *)webView shouldStartLoadWithRequest:(NS
NSURL *requestURL = request.URL;

if ([requestURL.host isEqualToString:@"localhost.de"]){
    // extract the query part
    NSString* queryPart = requestURL.query;
    NSDictionary *values = [NSURL ab_parseURLQueryString:queryPart]
    self.token = values[@"oauth_token"];
    self.verifier = values[@"oauth_verifier"];

    [self exchangeToken];

    return NO;
}

return YES;
}
```

As you can see, if the URL matches localhost.de, we extract the oauth_token and oauth_verifier from the request URL, trigger our exchangeToken-method which we'll discuss next and return NO to prevent the web view from actually loading the URL (it would be unsuccessful, since it doesn't exist.). If you're stepping through this code, you'll notice that the token returned here is the same one we passed in – we could eventually check for equality to make our implementation more robust, but it's not require and doesn't add any extra security on top.

Let's just sum up what we've done so far: Using a simple library to sign requests, making them OAuth-compatible, we requested a request token from Twitter. To do this, we had to sign the request using our apps' own consumer key and consumer secret. We then presented a website to our users we called using the just returned request token, identifying ourselves to Twitter. After the successful login, we intercept the redirect that is being performed automatically by Twitter (as a convention) and extract the oauth_verifier which we'll use in a second to exchange the request token for an access token. The access token can be used to make real requests for user data, something the request token is denied from.

So, the last step is to exchange the request token for an access token. The request setup should be familiar to you by now, since it's the same one we've used to perform the previous calls.

```
-(void)exchangeToken {
    NSMutableURLRequest *tokenExchange = [NSMutableURLRequest requestWi

    tokenExchange.HTTPMethod = @"POST";

    tokenExchange.HTTPBody = [[NSString stringWithFormat:@"oauth_verifi

    [tokenExchange addValue:OAuthorizationHeader(tokenExchange.URL, tok

    NSURLSessionDataTask *task = [self.session dataTaskWithRequest:toke
        if (data){
            NSString *payload = [[NSString alloc] initWithData:data enc
            NSDictionary *contentDictionary = [NSURL ab_parseURLQueryS

            self.token = contentDictionary[@"oauth_token"];
            self.tokenSecret = contentDictionary[@"oauth_token_secret"]

            [self retrieveTimeline];
        }
    }];
```

```
    [task resume];
}
```

The code snippet above should be self-explaining, I'll just add some remarks. We send the verifier we just got as the HTTP Post payload, add the Authorization-header as usually and then perform the request. If successful, the response contains the newly created request token and request token secret, both of which we need from now on. the last thing we do is to fetch the user's (your) timeline in the −retrieveTimeline-method.

```
-(void)retrieveTimeline {
    NSMutableURLRequest *request = [NSMutableURLRequest requestWithURL:

    [request addValue:OAuthorizationHeader(request.URL, request.HTTPMe

    NSURLSessionDataTask *task = [self.session dataTaskWithRequest:requ
        if (data){
            NSArray *tweets = [NSJSONSerialization JSONObjectWithData:d

            NSLog(@"%@",tweets);
        }

    }];

    [task resume];
}
```

My apologies for hardcoding the URL into the example, but code style besides, you should see a dump of a portion of your timeline in the console. Congratulations.

As you can see, once you've got the token and the according secret stored somewhere in your app, it's only a matter of adding the correct Authorization header to talk to an API that requires OAuth-authorization.

1.5 OAuth using a library

Reinventing the wheel is what we've done above, to demonstrate the complexity of the OAuth process and how to tackle it using only system frameworks.

If you need to work with OAuth for your app, consider using one of the existing libraries out there, like *OAuth2Client*, a project that handles the initial token retrieval flow and allows you to retrieve signed URL requests without having to add headers

yourself or other tedious stuff. This library can therefore be fully integrated to work with NSURLSession and friends.

1.6 HTTP Basic

Many applications rely on HTTP Basic authentication today. Good news is that the iOS-frameworks provides all means necessary for dealing with those servers, and it's really simple, too.

Conceptually, HTTP Basic and Digest rely on initially denying access to a resource using the 401 – Not authorized header and an additional header indicating *how* to authenticate. In the case of HTTP Basic, the server response might look something like

```
HTTP/1.1 401 Not Authorized
Date: Fri, 31 Dec 1999 23:59:59 GMT
Content-Type: text/html
Content-Length: 0
WWW-Authenticate: Basic realm="Admin Interface"
```

This is indication to the client that it's time to collect some credentials, in the case of HTTP Basic a username and a password, and to send it to the server in an attempt to access the protected resource.

A common yet wrong way to work with HTTP Basic authentication is to construct the authorization-header manually. This is not required, iOS is perfectly able to construct those headers for us, and even cache the Authorization and reuse it – either session wide or globally. As always, NSURLSession is the basic building block for this sample.

Once again, create an Single View Application from the Xcode templates. We'll work in the created view controller for this sample.

First thing we do is to create a session in the viewDidLoad: override.

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    NSURLSessionConfiguration *configuration = [NSURLSessionConfigurati

    self.session = [NSURLSession sessionWithConfiguration:configuration
}
```

We are using the ephemeral session configuration type here to avoid side effects that may impair this sample. Ephemeral sessions don't persist anything – neither

responses nor credentials.

We are also implementing the NSURLSessionTaskDelegate protocol in our view controller, specifically one method from the protocol. This is the method that handles the so-called authentication challenge.

```
- (void)URLSession:(NSURLSession *)session task:(NSURLSessionTask *

// Exit if the credentials were rejected more than 3 times.
if ([challenge previousFailureCount] > 3){
    NSLog(@"Three times unsuccessful, Exiting.");
    completionHandler(NSURLSessionAuthChallengeCancelAuthenticationChal
}

// Just handle HTTP Basic in this sample.
if ([challenge.protectionSpace.authenticationMethod isEqualToString:NS
    NSURLCredential *newCredentials = [NSURLCredential credentialWithUs
    completionHandler(NSURLSessionAuthChallengeUseCredential, newCrede
} else {
    completionHandler(NSURLSessionAuthChallengeCancelAuthenticationChal
}
}
```

What this method is doing is reacting to challenge requests from the framework, which in turn calls the delegate only if the server requires some form of authentication. The first 4 lines are handling the case where we tried to authenticate 3 times already and still haven't had success. To avoid staying there forever, we abort after three tries by calling the completionBlock indicating our final failure.

In the next part, we check if the authentication is in fact the part where we pass our credentials in and return them up to the framework. As you can see, this is actually quite simple.

One nice thing is the ability to control the scope of the credentials, i.e. being able to make those credentials valid permanently session wide or for this specific request only.

Now that everything is in place we can finally access a protected resource, like we've done a few times before.

```
-(void)viewDidAppear:(BOOL)animated {
    [super viewDidAppear:animated];

    NSURL *url = [NSURL URLWithString:@"http://test.webdav.org/auth-bas

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

```
    NSURLSessionDataTask *task = [self.session dataTaskWithRequest:requ
        if (data.length){
            NSString *content = [[NSString alloc] initWithData:data enc
            NSLog(@"Success! %@", content);
        }
    }];

    [task resume];
}
```

That simple. The best part really is that after we've set up the delegate correctly, we can just use the tasks as always, they need no modification in any way.

1.7 One last word

One thing to point out here is that you have to use one of the lower-level APIs to request protected resources, be it OAuth or HTTP-Basic/Digest, since the dataWithURL and friends cannot be customized to work with OAuth – and even if it was possible, sometimes control is better than ease of use.