

---

**NOTE:** This repository is no longer supported or updated by GitHub. If you wish to continue to develop this code yourself, we recommend you fork it.

## OctoKit

Carthage compatible

OctoKit is a Cocoa and Cocoa Touch framework for interacting with the GitHub API, built using AFNetworking, Mantle, and ReactiveCocoa.

### Making Requests

In order to begin interacting with the API, you must instantiate an `OCTClient`. There are two ways to create a client without authenticating:

1. `-initWithServer:` is the most basic way to initialize a client. It accepts an `OCTServer`, which determines whether to connect to GitHub.com or a GitHub Enterprise instance.
2. `+unauthenticatedClientWithUser:` is similar, but lets you set an *active user*, which is required for certain requests.

We'll focus on the second method, since we can do more with it. Let's create a client that connects to GitHub.com:

```
1 OCTUser *user = [OCTUser userWithRawLogin:username server:OCTServer.  
    dotComServer];  
2 OCTClient *client = [OCTClient unauthenticatedClientWithUser:user];
```

After we've got a client, we can start fetching data. Each request method on `OCTClient` returns a ReactiveCocoa signal, which is kinda like a future or promise:

```
1 // Prepares a request that will load all of the user's repositories,  
    represented  
2 // by `OCTRepository` objects.  
3 //  
4 // Note that the request is not actually _sent_ until you use one of  
    the  
5 // -...subscribe methods below.  
6 RACSignal *request = [client fetchUserRepositories];
```

However, you don't need a deep understanding of RAC to use OctoKit. There are just a few basic operations to be aware of.

---

## Receiving results one-by-one

It often makes sense to handle each result object independently, so you can spread any processing out instead of doing it all at once:

```
1 // This method actually kicks off the request, handling any results
  // using the
2 // blocks below.
3 [request subscribeNext:^(OCTRepository *repository) {
4     // This block is invoked for _each_ result received, so you can
      // deal with
5     // them one-by-one as they arrive.
6 } error:^(NSError *error) {
7     // Invoked when an error occurs.
8     //
9     // Your `next` and `completed` blocks won't be invoked after this
      // point.
10 } completed:^(
11     // Invoked when the request completes and we've received/processed
      // all the
12     // results.
13     //
14     // Your `next` and `error` blocks won't be invoked after this point
      // .
15 }];
```

## Receiving all results at once

If you can't do anything until you have *all* of the results, you can “collect” them into a single array:

```
1 [[request collect] subscribeNext:^(NSArray *repositories) {
2     // Thanks to -collect, this block is invoked after the request
      // completes,
3     // with _all_ the results that were received.
4 } error:^(NSError *error) {
5     // Invoked when an error occurs. You won't receive any results if
      // this
6     // happens.
7 }];
```

## Receiving results on the main thread

The blocks in the above examples will be invoked in the background, to avoid slowing down the main thread. However, if you want to run UI code, you shouldn't do it in the background, so you must “deliver” results to the main thread instead:

---

```

1  [[request deliverOn:RACScheduler.mainThreadScheduler] subscribeNext:^(
    OCTRepository *repository) {
2      // ...
3  } error:^(NSError *error) {
4      UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Whoops!"
5                                                              message:@"Something
                                                                went wrong."
6                                                              delegate:nil
7                                                              cancelButtonTitle:nil
8                                                              otherButtonTitles:nil];
9      [alert show];
10 } completed:^(
11     [self.tableView reloadData];
12 }];

```

## Cancelling a request

All of the `-subscribe...` methods actually return a `RACDisposable` object. Most of the time, you don't need it, but you can hold onto it if you want to cancel requests:

```

1  - (void)viewWillAppear:(BOOL)animated {
2      [super viewWillAppear:animated];
3
4      RACDisposable *disposable = [[[[self.client
5          fetchUserRepositories]
6          collect]
7          deliverOn:RACScheduler.mainThreadScheduler]
8          subscribeNext:^(NSArray *repositories) {
9              [self addTableViewRowsForRepositories:repositories];
10         } error:^(NSError *error) {
11             UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"
12                                     Whoops!"
13                                                         message:@"
14                                                         Something
15                                                         went
16                                                         wrong."
17                                                         delegate:nil
18                                                         cancelButtonTitle:nil
19                                                         otherButtonTitles:nil
20                                                         ];
21             [alert show];
22         }];
23
24     // Save the disposable into a `strong` property, so we can access
25     // it later.
26     self.repositoriesDisposable = disposable;
27 }

```

---

```
23 - (void)viewWillDisappear:(BOOL)animated {
24     [super viewWillDisappear:animated];
25
26     // Cancels the request for repositories if it's still in progress.
27     // If the
28     // request already terminated, nothing happens.
29     [self.repositoriesDisposable dispose];
30 }
```

## Authentication

OctoKit supports two variants of OAuth2 for signing in. We recommend the browser-based approach, but you can also implement a native sign-in flow if desired.

In both cases, you will need to register your OAuth application, and provide OctoKit with your client ID and client secret before trying to authenticate:

```
1 [OCTClient setClientID:@"abc123" clientSecret:@"654321abcdef"];
```

## Signing in through a browser

With this API, the user will be redirected to their default browser (on OS X) or Safari (on iOS) to sign in, and then redirected back to your app. This is the easiest approach to implement, and means the user never has to enter their password directly into your app — plus, they may even be signed in through the browser already!

To get started, you must implement a custom URL scheme for your app, then use something matching that scheme for your OAuth application's callback URL. The actual URL doesn't matter to OctoKit, so you can use whatever you'd like, just as long as the URL scheme is correct.

Whenever your app is opened from your URL, or asked to open it, you must pass it directly into `OCTClient`:

```
1 - (BOOL)application:(UIApplication *)application openURL:(NSURL *)URL
  sourceApplication:(NSString *)sourceApplication annotation:(id)
  annotation {
2     // For handling a callback URL like my-app://oauth
3     if ([URL.host isEqualToString:@"oauth"]) {
4         [OCTClient completeSignInWithCallbackURL:URL];
5         return YES;
6     } else {
7         return NO;
8     }
9 }
```

---

After that's set up properly, you can present the sign in page at any point. The pattern is very similar to making a request, except that you receive an `OCTClient` instance as a reply:

```
1  [[OCTClient
2      signInToServerUsingWebBrowser:OCTServer.dotComServer scopes:
3          OCTClientAuthorizationScopesUser]
4      subscribeNext:^(OCTClient *authenticatedClient) {
5          // Authentication was successful. Do something with the created
6          client.
7      } error:^(NSError *error) {
8          // Authentication failed.
9      }];
```

You can also choose to receive the client on the main thread, just like with any other request.

### Signing in through the app

If you don't want to open a web page, you can use the native authentication flow and implement your own sign-in UI. However, two-factor authentication makes this process somewhat complex, and the native authentication flow may not work with GitHub Enterprise instances that use single sign-on.

Whenever the user wants to sign in, present your custom UI. After the form has been filled in with a username and password (and perhaps a server URL, for GitHub Enterprise users), you can attempt to authenticate. The pattern is very similar to making a request, except that you receive an `OCTClient` instance as a reply:

```
1  OCTUser *user = [OCTUser userWithRawLogin:username server:OCTServer.
2      dotComServer];
3  [[OCTClient
4      signInAsUser:user password:password oneTimePassword:nil scopes:
5          OCTClientAuthorizationScopesUser]
6      subscribeNext:^(OCTClient *authenticatedClient) {
7          // Authentication was successful. Do something with the created
8          client.
9      } error:^(NSError *error) {
10         // Authentication failed.
11     }];
```

*(You can also choose to receive the client on the main thread, just like with any other request.)*

`oneTimePassword` must be `nil` on your first attempt, since it's impossible to know ahead of time if a user has two-factor authentication enabled. If they do, you'll receive an error of code `OCTClientErrorTwoFactorAuthenticationOneTimePasswordRequired`, and should present a UI for the user to enter the 2FA code they received via SMS or read from an authenticator app.

---

Once you have the 2FA code, you can attempt to sign in again. The resulting code might look something like this:

```
1 - (IBAction)signIn:(id)sender {
2     NSString *oneTimePassword;
3     if (self.oneTimePasswordVisible) {
4         oneTimePassword = self.oneTimePasswordField.text;
5     } else {
6         oneTimePassword = nil;
7     }
8
9     NSString *username = self.usernameField.text;
10    NSString *password = self.passwordField.text;
11
12    [[[OCTClient
13        signInAsUser:username password:password oneTimePassword:
14        oneTimePassword scopes:OCTClientAuthorizationScopesUser]
15        deliverOn:RACScheduler.mainThreadScheduler]
16        subscribeNext:^(OCTClient *client) {
17            [self successfullyAuthenticatedWithClient:client];
18        } error:^(NSError *error) {
19            if ([error.domain isEqual:OCTClientErrorDomain] && error.
20                code ==
21                OCTClientErrorTwoFactorAuthenticationOneTimePasswordRequired
22            ) {
23                // Show OTP field and have the user try again.
24                [self showOneTimePasswordField];
25            } else {
26                // The error isn't a 2FA prompt, so present it to the
27                user.
28                [self presentError:error];
29            }
30        }];
31    }
```

### Choosing an authentication method dynamically

If you really want a native login flow without sacrificing the compatibility of browser-based login, you can inspect a server's metadata to determine how to authenticate.

However, because not all GitHub Enterprise servers support this API, you should handle any errors returned:

```
1 [[OCTClient
2     fetchMetadataForServer:someServer]
3     subscribeNext:^(OCTServerMetadata *metadata) {
4         if (metadata.supportsPasswordAuthentication) {
```

---

```
5         // Authenticate with +signInAsUser:password:oneTimePassword
           :scopes:
6     } else {
7         // Authenticate with +signInToServerUsingWebBrowser:scopes:
8     }
9     } error:^(NSError *error) {
10         if ([error.domain isEqual:OCTClientErrorDomain] && error.code
            == OCTClientErrorUnsupportedServer) {
11             // The server doesn't support capability checks, so fall
              back to one
12             // method or the other.
13         }
14     }];
```

## Saving credentials

Generally, you'll want to save an authenticated OctoKit session, so the user doesn't have to repeat the sign in process when they open your app again.

Regardless of the authentication method you use, you'll end up with an `OCTClient` instance after the user signs in successfully. An authenticated client has `user` and `token` properties. To remember the user, you need to save `user.rawLogin` and the OAuth access token into the keychain.

When your app is relaunched, and you want to use the saved credentials, skip the normal sign-in methods and create an authenticated client directly:

```
1 OCTUser *user = [OCTUser userWithRawLogin:savedLogin server:OCTServer.
   dotComServer];
2 OCTClient *client = [OCTClient authenticatedClientWithUser:user token:
   savedToken];
```

If the credentials are still valid, you can make authenticated requests immediately. If not valid (perhaps because the OAuth token was revoked by the user), you'll receive an error after sending your first request, and can ask the user to sign in again.

## Importing OctoKit

OctoKit is still new and moving fast, so we may make breaking changes from time-to-time, but it has partial unit test coverage and is already being used in GitHub for Mac's production code.

To add OctoKit to your application:

1. Add the OctoKit repository as a submodule of your application's repository.
2. Run `script/bootstrap` from within the OctoKit folder.

- 
3. Drag and drop `OctoKit.xcodeproj`, `OctoKitDependencies.xcodeproj`, `ReactiveCocoa.xcodeproj`, and `Mantle.xcodeproj` into the top-level of your application's project file or workspace. The latter three projects can be found within the `External` folder.
  4. On the “Build Phases” tab of your application target, add the following to the “Link Binary With Libraries” phase:
    - **On iOS**, add the `.a` libraries for OctoKit, AFNetworking, and ISO8601DateFormatter.
    - **On OS X**, add the `.framework` bundles for OctoKit, ReactiveCocoa, Mantle, AFNetworking, and ISO8601DateFormatter. All of the frames must also be added to any “Copy Frameworks” build phase.
  5. Add `$(BUILD_ROOT)/../IntermediateBuildFilesPath/UninstalledProducts/include $(herited)` to the “Header Search Paths” build setting (this is only necessary for archive builds, but it has no negative effect otherwise).
  6. **For iOS targets**, add `-ObjC` to the “Other Linker Flags” build setting.

If you would prefer to use CocoaPods, there are some OctoKit podspecs that have been generously contributed by third parties.

If you're developing OctoKit on its own, then use `OctoKit.xcworkspace`.

## Copying the frameworks

*This is only needed **on OS X**.*

1. Go to the “Build Phases” tab of your application target.
2. If you don't already have one, add a “Copy Files” build phase and target the “Frameworks” destination.
3. Drag `OctoKit.framework` from the OctoKit project's `Products` Xcode group into the “Copy Files” build phase you just created (or the one that you already had).
4. A reference to the framework will now appear at the top of your application's Xcode group, select it and show the “File Inspector”.
5. Change the “Location” to “Relative to Build Products”.
6. Now do the same (starting at step 2) for the frameworks within the External folder.

## License

OctoKit is released under the MIT license. See `LICENSE.md`.