Legacy - iOS Integration Reference

This document will show you how to integrate the Alchemer Mobile iOS SDK into your app, configure it, and test to make sure it's working properly. Each section lists the minimum necessary configuration, as well as optional steps.

System Requirements

Minimum Deployment Target: iOS 10.3

Minimum Xcode Version: Xcode 9.0

Dependencies

The Alchemer Mobile iOS SDK has no external dependencies. The SDK is written entirely in Objective-C, but swift-friendly method names are provided where appropriate.

Please note that we do not currently support SwiftUI. If you'd like us to build support in the future, please let us know.

SDK Size

- Our SDK is very small. It will add approximately **1.2 MB** to the size of your app.

Supported Languages

We have translated all hard-coded strings in our SDK into the following languages. The content of all Interactions comes from our server, and you may translate the text for each Interaction by visiting the Translations Page.

Please note that you must also specify the target language in your Xcode project by selecting the project > clicking Info > then, under Localizations, click the Add button for the desired language.

Locale Qualifier	Language Name
None	English
`ar`	Arabic
`el`	Greek
`da`	Danish
`de`	German
`es`	Spanish
`fr`	French

Locale Qualifier	Language Name
`fr-CA`	French Canadian
`it`	Italian
`ja`	Japanese
`ko`	Korean
`nl`	Dutch
`pl`	Polish
`pt-BR`	Brazilian Portuguese
`ru`	Russian
`sv`	Swedish
`tr`	Turkish
`zh-Hant`	Chinese (Traditional)
`zh-Hans`	Chinese (Simplified)

Adding Alchemer Mobile

> To ensure the SDK functions properly, when upgrading please make sure to read the Migration Guide for each version above the version you are currently using.

There are three supported methods to add the Alchemer Mobile SDK to your project: CocoaPods, Carthage, and subproject.

CocoaPods

Using the CocoaPods dependency manager is usually the easiest way to integrate. If you haven't already, you will need to install CocoaPods on your development machine.

If you haven't used CocoaPods with your project yet, you will need to create a so-called **Podfile**. The easiest way to do this is to open up Terminal, change to your project directory, and run `pod init`.

Next you'll want to open your podfile in a text editor and add the `apptentive-ios` pod to the list of pods for your target.

```
 platform :ios, '9.0'
use_frameworks!

target '<name of your target>' do
  ...
  pod 'apptentive-ios'
  ...
end
```

Then run `pod install`. When it finishes, open up the workspace file that CocoaPods created.

Carthage

Using Carthage is another simple way of integrating Alchemer Mobile. If you haven't already, you will need to install Carthage on your development machine.

If you haven't used Carthage with your project yet, you will need to create a so-called **Cartfile**. This is a plain text file that goes in the same directory as your project's `.xcodeproj` file.

Create or open this file in a text editor and add the `apptentive-ios` framework to it as follows:

```
github 'apptentive/apptentive-ios' >= 5.0.0
```

Then run `carthage update`. This will fetch the Alchemer Mobile SDK into the Carthage/Checkouts folder and build it.

When the command finishes, open your app's project in Xcode and select your project file in the Project Navigator. Switch to the "General" settings tab and scroll down to "Linked Frameworks and Libraries". Then open the Carthage/Build folder (in the same parent folder as your Cartfile) in a Finder window. Drag the `Apptentive.framework` file from the Finder window and drop it into the Linked Frameworks and Libraries section in the Xcode window (note that you should not add the `ApptentiveDebugging.framework` to your project).

The final step is to add a Run Script build phase to your project to appropriately package the included framework(s) for development and distribution. Switch to the "Build Phases" settings tab, click the "+" icon, and choose "New Run Script Build Phase". Paste in the following as the body of the Run Script build phase (the shell should remain set to `/bin/sh`):

```
/usr/local/bin/carthage copy-frameworks
```

Then scroll down to the "Input Files" field and add the following:

```
$(SRCROOT)/Carthage/Build/iOS/Apptentive.framework
```

Finally scroll down to the "Output Files" field and add the following:

```
$(DERIVED_FILE_DIR)/Apptentive.framework
```

Subproject

If you prefer to not use a package manager, you can integrate the Alchemer Mobile SDK as a subproject of your app project.

Start by cloning the apptentive-ios repository on Github:

```
$ git clone https://github.com/apptentive/apptentive-ios.git
```

If you are using git for your app project, you can also add the Alchemer Mobile iOS SDK as a submodule:

```
$ git submodule add https://github.com/apptentive/apptentive-ios.git apptentive-ios
```

Then open your app project in Xcode and switch to the Project Navigator, and open the Alchemer Mobile folder (in the root of the apptentive-ios clone) in a Finder window. Drag the Apptentive.xcodeproj file from the Finder window into the Xcode Project Navigator below your app project.

Next, select your app project in the Project Navigator and switch to the “Build Phases” tab. Expand the “Target Dependencies” section and click the “+” button. Choose the Alchemer Mobile framework from the list and click the “Add” button. Repeat this for the “Embedded Binaries” section under the “General” tab.

Initialize Alchemer Mobile

When your app starts, it will need to initialize the Alchemer Mobile SDK. You will need to provide your Alchemer Mobile App Key and Alchemer Mobile App Signature, available in the API & Development section under the Settings tab of the Alchemer Mobile Dashboard, along with the identifier for your app in the App Store (available in iTunes Connect by clicking My Apps, choosing the app, and copying the number under “Apple ID”).

You will need to import the Alchemer Mobile module and create an instance of the `ApptentiveConfiguration` class with your Alchemer Mobile App Key and Alchemer Mobile App Signature. Then call the `Apptentive.register(with:)` method. We recommend registering Alchemer Mobile in your application delegate’s `application(_:didFinishLaunchingWithOptions:)` method:

```
 import UIKit
import Apptentive

class AppDelegate: UIResponder, UIApplicationDelegate {
var window: UIWindow?

  func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions: [UIApplicationL
aunchOptionsKey: Any]?) -> Bool {
    if let configuration = ApptentiveConfiguration(apptentiveKey: "<#Your Apptentive App Key#>", apptentive
Signature: "<#Your Apptentive App Signature>") {
        configuration.appID = "<#Your iTunes App ID#>"
        // Set the log level to Verbose to debug Apptentive. Default is Info level.
        configuration.logLevel = ApptentiveLogLevelVerbose;
        Apptentive.register(with: configuration)
    }


    // Other app initialization...

    return true
  }
}
```

> The `register` method must be called on the main thread. Internally it will run operations on a background thread wherever possible and will not perform any long-running operations on the main thread.

> Make sure you use the Alchemer Mobile App Key and Alchemer Mobile App Signature for the iOS app you created in the Alchemer Mobile console. Sharing these in two apps, or using ones from the wrong platform is not supported, and will lead to incorrect behavior.

Message Center

> See: How to Use Message Center

With the **Alchemer Mobile Message Center** your customers can send feedback, and you can reply, all without making them leave the app. Handling support inside the app will increase the number of support messages received and ensure a better customer experience.

Message Center lets customers see all the messages they have sent you, read all of your replies, and even send screenshots that may help debug issues.

Showing Message Center

Find a place in your app for a button that will launch Message Center. This will allow customers to contact you with feedback, or questions if they are having trouble using your app, as well as allow them to see your responses.

If Message Center is available, show a `UIButton` or `UITableViewCell` that will launch it when tapped. This example assumes you have an `UIViewController` subclass called `SettingsViewController` that has a `UIButton` you would like to open Message Center with.

```
 import UIKit
import Apptentive

class SettingsViewController: UIViewController {

  // ...

  @IBAction func openMessageCenter(sender: UIButton) {
    Apptentive.shared.presentMessageCenter(from: self)
  }
}
```

Checking Unread Message Count

You can also check to see how many messages are waiting to be read in the customer's Message Center.

```
Apptentive.shared.unreadMessageCount
```

Unread Message Count Notification

Your app can register for the `ApptentiveMessageCenterUnreadCountChanged` notification to help inform users that they have unread messages waiting:

```
 class ViewController: UIViewController {
  override func viewDidLoad() {
    super.viewDidLoad()

    NotificationCenter.default.addObserver(self, selector: #selector(ViewController.unreadMessageCountChan
ged), name: Notification.Name.ApptentiveMessageCenterUnreadCountChanged, object: nil)
  }

  @objc func unreadMessageCountChanged(notification: Notification) {
    if let count = notification.userInfo?["count"] as? Int {
      // update UI with count
    }
  }
}
```

You can use this notification to set your app's badge with the unread message count.

Alternatively, the Alchemer Mobile (Formerly Apptentive) class returns a `UIView` instance from the `unreadMessageCountAccessoryView(apptentiveHeart:)` method that will update itself and is ideal for use as the accessory view in a `UITableViewCell`.

Attachments

Attachments are messages that you can send from the SDK programmatically, which will be visible to you in the Conversation View, but will not be visible to your customers in Message Center. They are great for sending contextual information or logs from rare crash events.

Hidden File Attachments

- sendAttachment(_:mimeType:)

Hidden Image Messages

- sendAttachment(image:)

Hidden Text Messages

- sendAttachment(text:)

Events

Events record user interaction. You can use them to determine if and when an Interaction will be shown to your customer. You will use these Events later to target Interactions, and to determine whether an Interaction can be shown. You trigger an Event with the `engage()` method. This will record the Event, and then check to see if any Interactions targeted to that Event are allowed to be displayed, based on the logic you set up in the Alchemer Mobile Dashboard.

One good place to engage an event is when a view controller appears, for example in the view controller's `viewDidAppear(_:)` method:

```
override func viewDidAppear(animated: Bool) {
  super.viewDidAppear(animated)

  // Engage the "viewed_list" event.
  Apptentive.shared.engage(event: "viewed_list", from: self)
}
```

Another good place to engage events is from the action methods in your view controllers:

```
@IBAction func likeArticle(_ sender: AnyObject?) {
  // ...

  // Engage the "liked_article" event.
  Apptentive.shared.engage(event: "liked_article", from: self)
}
```

If you want to engage an event when a modal view is dismissed, you will want to call the engage method in the completion block, and pass the presenting view controller as the `from` parameter:

```
@IBAction func save(_ sender: AnyObject?) {
  // save the item...

  self.dismiss(animated: true) {
    // Engage the "saved_item" event.
    Apptentive.shared.engage(event: "saved_item", from: self.presentingViewController)
  }
}
```

This ensures that the view controller you pass in will still be visible if the event you engage results in an interaction being displayed.

You can also adopt the delegate protocol for a tab bar or navigation controller to keep track of

when the user changes tabs or pops and pushes view controllers:

```
func tabBarController(_ tabBarController: UITabBarController, didSelect viewController: UIViewController) {
    if tabBarController.viewControllers?.index(of: viewController) ?? 0 == 0 {
        // Engage the "photos_tab_selected" event.
        Apptentive.shared.engage(event: "photos_tab_selected", from: tabBarController)
    } else {
        // Engage the "favorites_tab_selected" event.
        Apptentive.shared.engage(event: "favorites_tab_selected", from: tabBarController)
    }
}
```

Finally, you can engage an event when your app encounters an error:

```
do {
    try context.save()
} catch let error as NSError {
    print("Error saving context: \(error)")

    // Engage the "core_data_save_failed" event.
    Apptentive.shared.engage(event: "core_data_save_failed", from: self)
}
```

This may allow you to let users know if there is a workaround or app update that fixes the problem.

> We recommend that you create at least 10 Events. This gives you the flexibility to choose the best place for Interactions to display after your app is live, without having to update your app.

Event Names

Our web dashboard works best for up to several dozen unique event names. It does not work well if you auto-generate thousands of unique event names. We suggest using spaces, underscores, or dashes between words in Event names, and avoiding any other special characters. If you plan to target users based on viewing a piece of content out of a collection of hundreds or thousands (say, SKUs in an online retail app), do not create event names for each piece of content. Instead, you can use Custom Person Data for item viewed.

For example, you could set a key of `viewed_item` with a value `123456`. You could then target users for whom that key matches, or is not null.

> #### Custom Data
> Do not use engage:withCustomData to send Custom Data to Alchemer Mobile. Refer to the Custom Data section to properly send it.

Interactions

All of the following Interactions can be configured in the Alchemer Mobile Dashboard to show up

when any of your Events are engaged.

Love Dialog

Love Dialogs can help learn about your customer, asking customers that love your app to rate it in the applicable app store, and customer who don't love it yet to give you feedback, or answer a Survey.

> See: How to Use Ratings Prompts

Surveys

Surveys are a powerful tool for learning about your customers' needs.

> See: How to Use Surveys

Survey Finished Notification

You can be notified of a survey being completed by listening for a `ApptentiveSurveySentNotification` using `NotificationCenter`:

```
override func viewDidLoad() {
 super.viewDidLoad()

 NotificationCenter.default.addObserver(self, selector: #selector(surveyFinished), name: NSNotification.Nam
e.ApptentiveSurveySentNotification, object: nil)
}
func surveyFinished(notification: Notification) {
 // Called when the user submits a survey
}
```

Prompts (formerly Notes)

Prompts allow you to show an alert to customers, and optionally direct them to a Survey, Message Center, a Deep Link, or simply dismiss the Prompt.

> See: How to Use Prompts

Push Notifications

Alchemer Mobile can send push notifications to your app when you reply to your customers in Message Center, which will give your customers an indication that you have replied to them along with guiding them back into your app and into Message Center.

To set up push notifications, you will need to make a few small modifications to your application's App Delegate object, enable push notifications in your app, and add your push credentials to the Integrations page of your Alchemer Mobile dashboard.

Configuring Your Application Delegate for Push

There are a few additions or changes you will have to make to your application's App Delegate class. If your app was not previously set up to register for and receive push notifications, you will have to add a bit of code that runs at app launch and implement a handful of methods to register for and respond to notifications:

- In your `application(_:didFinishLaunchingWithOptions:)` method, call the `registerForRemoteNotifications()` method on the application object. If your app already uses push notifications, this method should already be called, so you can skip this step.
- In the same method, set the Alchemer Mobile SDK as the delegate for `UNUserNotificationCenter` as follows:

```
UNUserNotificationCenter.current().delegate = Apptentive.shared
```

- If your app has another object acting as the user notification center's delegate, you need to implement two methods in that object to pass these calls on to the Alchemer Mobile SDK:

```
@objc func userNotificationCenter(_ center: UNUserNotificationCenter, didReceive response: UNNotificationR
esponse, withCompletionHandler completionHandler: @escaping () -> Void) {
  let handledByApptentive = Apptentive.shared.didReceiveNotificationResponse(response, from:
self.window?.rootViewController, withCompletionHandler: completionHandler)
  if (!handledByApptentive) {
    // Pass the notification on to your code and be sure to call the completion handler when finished.
    completionHandler()
  }
}

func userNotificationCenter(_ center: UNUserNotificationCenter, willPresentNotification: UNNotification, withC
ompletionHandler completionHandler: @escaping (UNNotificationPresentationOptions) -> Void) {
  let handledByApptentive = Apptentive.shared.willPresentNotification(notification, withCompletionHandler: c
ompletionHandler)
  if (!handledByApptentive) {
    // Pass the notification on to your code and be sure to call the completion handler when finished.
    completionHandler([])
  }
}
```

- Implement the `application(_:didRegisterForRemoteNotificationsWithDeviceToken:)` method as follows:

```
@objc func application(_ application: UIApplication, didRegisterForRemoteNotificationsWithDeviceToken devi
ceToken: Data) {
  Apptentive.shared.setPushProvider(.apptentive, deviceToken: deviceToken)
}
```

- (Optional) Implement the `application(_:didFailToRegisterForRemoteNotificationsWithError:)` method as follows:

```
@objc func application(_ application: UIApplication, didFailToRegisterForRemoteNotificationsWithError error:
Error) {
  // Log the failure to register to help with debugging, e.g.:
  print("Failed to register for remote notifications with error: \(error)");
}
```

- Implement the `application(_:didReceiveRemoteNotification: fetchCompletionHandler:)` method as follows:

```
 @objc func application(_ application: UIApplication, didReceiveRemoteNotification userInfo: [AnyHashable : A
ny], fetchCompletionHandler completionHandler: @escaping (UIBackgroundFetchResult) -> Void) {
  let handledByApptentive = Apptentive.shared.didReceiveRemoteNotification(userInfo, fetchCompletionHand
ler: completionHandler)
  if !handledByApptentive {
    // handle non-Apptentive push notification if needed
    completionHandler(.noData)
  }
}
```

- Finally, you will have to request permission from the user to display alerts and play sounds. You can either do this right at launch (in `application(_:didFinishLaunchingWithOptions:)`), or upon receiving a `ApptentiveMessageSentNotification`. To display the permission dialog the first time that the user sends a message, add the following code:

```
 NotificationCenter.default.addObserver(forName: NSNotification.Name.ApptentiveMessageSent, object: nil, q
ueue: OperationQueue.main) {
  (notification) in UNUserNotificationCenter.current().requestAuthorization(options: [.alert, .sound]) {_,_ in }
}
```

To display the permission request immediately at launch, simply call the `requestAuthorization(options:completionHandler:)` method without enclosing it in a notification handler.

Provisioning Your App For Push

1. If your app is not yet configured to receive push notifications, start by selecting your app project in the project navigator, selecting your app target in the target list, and selecting the Capabilities tab.You will have to turn on both the Background Modes and Push Notifications sections. In the Background Modes section, check the box next to Remote Notifications. In the Push Notifications section, follow any instructions to enable push for your app ID.

2. In the Apple Developer Portal's Certificates, Identifiers and Profiles section, configure your app's App ID for push. To do this, switch to the App IDs section, select the ID for your app, and click the Edit button. Scroll down to the Push Notification section (which should be enabled after completing the step above), and click the Create Certificate button for the certificates you need for testing or deploying your app (we recommend creating certificates for both the Development—sometimes called "sandbox"—and Production environments). Follow the instructions, and when you are finished, you should have downloaded a certificate for each environment.

3. In Finder, open each certificate to add it to your login keychain.

Uploading your Push Credentials to the Alchemer Mobile Dashboard

1. Launch the Keychain Access app on your Mac (in /Applications/Utilities) and choose the My Certificates category in the "login" keychain. You will see certificates with names like "Apple Production IOS Push Services: com.my.app.id" (where com.my.app.id is your app's bundle

identifier) and “Apple Development IOS Push Services: com.my.app.id”. For each certificate, click the disclosure triangle next to it and shift-click to select both the certificate and its key. Then choose File > Export Items.... You will need to encrypt the exported certificate and private key with a password of your choosing, and export it using the PKCS-12 (.p12) format.

2. Go to the Integrations page of your Alchemer Mobile dashboard for your app in your browser, and expand the Alchemer Mobile Push section. Enter the password you chose when exporting your .p12 file into the Push Certificate Password field, and drag the .p12 file into the Push Certificate field. You can choose to enable a notification sound, and even specify the name of an audio file in your app’s bundle to play in place of the default sound. Then click the Save button and switch on the Active toggle.

Testing Push

At this point you should be able to run your app on a device, send a message in Message Center, and reply in your Alchemer Mobile dashboard with the app either open or closed, and receive a push notification banner on your device. Tapping the banner should open your app if it is not already running and present the Alchemer Mobile Message Center view controller.

You can also test your app using the production push environment by archiving your app and uploading it to a service such as Microsoft App Center or TestFlight.

It typically takes roughly a minute for a push notification to be received by the device after replying to a message in the dashboard.

If these tests fail, please see our troubleshooting guide for more information, or feel free to reach out to our customer success team.

Customer Information

Set Customer Contact Information

If you already know the customer’s email address or name, you can pass them to us to display in the conversation view on your Alchemer Mobile dashboard.

```
Apptentive.shared.personEmailAddress = <#Email Address#>
```

```
Apptentive.shared.personName = <#Person Name#>
```

Message Center provides dialogs that allow your customers to set their name and email as well. Calling the above methods will overwrite what your customer enters. If you don’t want to overwrite what they enter, you can check their values first, using `Apptentive.shared.personEmailAddress` and `Apptentive.shared.personName`.

Custom Data

You can send Custom Data associated with a person’s profile that is using the app, or the device. In

particular, this is useful for sending a Customer ID and other information that helps you understand and support your users better. Custom Data can also be used for configuring when Interactions will run. You can add custom data of type `String`, `Number`, and `Boolean`.

In general, Custom Data can be sent as Person Custom Data or Device Custom Data. However, if sending a Customer ID, you must send it as Person Custom Data. For more on the benefits of setting a Customer ID, see here.

After the Custom Data field has been triggered, it will appear on the targeting screen for any Interaction within a few minutes. You may need to refresh your browser to see recent changes.

```
 Apptentive.shared.addCustomPersonData("1234321", withKey: "CustomerID")
Apptentive.shared.addCustomPersonData("Seattle", withKey: "city")
Apptentive.shared.addCustomPersonData(500, withKey: "points")
Apptentive.shared.addCustomPersonData(true, withKey: "is_premium")

Apptentive.shared.addCustomDeviceData("test@apptentive.com", withKey: "primary_account")
Apptentive.shared.addCustomDeviceData(5, withKey: "user_count")
Apptentive.shared.addCustomDeviceData(false, withKey: "full_version")
```

Customer Authentication

If you have multiple customers using your app, you may want to use Customer Authentication to protect each customer's information from one another. Customer Authentication requires that you have authentication built into your app, and will also require you to modify your server's authentication code to pass authentication information back to your app, and then to Alchemer Mobile. For more information on this feature, see our Customer Authentication Configuration.

If you do not want to use Customer Authentication, or don't have a login/authentication mechanism in your app, then Alchemer Mobile will still function, but all information will be stored in the same conversation.

How we log a customer in

Your server will authenticate a customer when they log in. At that time, you will need to generate a JSON Web Token (JWT) with a specific format, and signed with the JWT Signing Secret in your app's API & Development page.

Logging a Customer In

The JWT will be a string. When your server generates a JWT, you will need to send it back to your app, and then log in to Alchemer Mobile with it. You will also need to pass in a callback that will allow you to handle login failures. Your callback must implement the Apptentive.LoginCallback interface.

> Alchemer Mobile will securely manage the JWT. It is important not to reuse a JWT, or to store it in the app.

```
Apptentive.shared.logIn(token: customerJwt) { success, error in
    // Handle login success or failure here.
}
```

Logging a Customer Out

You should make sure to log a customer out any time you invalidate the customers session in your app. That means that when a customer explicitly logs out, you should also log them out of Alchemer Mobile. When they are logged out after a certain amount of time, you should likewise also log them out of Alchemer Mobile.

```
Apptentive.shared.logOut()
```

Note: If your customer has logged out of your app, but you don't log them out of Alchemer Mobile, their personal information may be visible to other app users.

Handling Authentication Failures

The JWT you create will have an expiration date, and will be signed with a secret. When the JWT expires, the server will reject any requests made with it. In this case, you should ask your customer to log in again. Other failure reasons are provided as well, but are only likely to occur during integration if there is a mistake in how you generate a JWT.

It is a good practice to choose an expiration that is longer than your normal session duration so that Alchemer Mobile does not cause your customer to need to re-authenticate.

```
Apptentive.shared.authenticationFailureCallback = { (reason, message) in
        // Handle authentication failure here.
    }
```

Logged Out Experience

When no customer is logged in, Alchemer Mobile public API methods will no-op. If you are using Message Center, and the button that launches it is visible in a part of your app that your customers can access without logging in to your app, you should follow the Message Center instructions above to hide the button unless Message Center can be shown.

Other

Customizing the Look and Feel

Please see our Customization Guide for more information.

Logging

The Alchemer Mobile SDK has four logging levels: Debug, Info, Warning, and Error. All logging levels are activated in debug builds when you integrate using CocoaPods or as an Xcode subproject. In release builds, and when integrating via a method that doesn't compile the SDK from source, only Info, Warning and Error log messages are enabled.

Related Articles