Xamarin.iOS Plugin

> Microsoft is replacing Xamarin with .Net MAUI. Starting with Alchemer Digital SDK version 6.7, new features will no longer be offered for the Xamarin plugin. You can reference our MAUI documentation here.

This document will show you how to integrate the Alchemer Mobile (Apptentive) SDK for Xamarin.iOS 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 11.0

Minimum Xcode Version: Xcode 13.0

Dependencies

The Alchemer Mobile (Apptentive) Xamarin SDK for iOS has no external dependencies

SDK Size

- Our SDK is very small. It will add approximately **2.3 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.

Locale Qualifier	Language Name
None	English
`ar`	Arabic
`el`	Greek
`da`	Danish
`de`	German
`es`	Spanish
`fr`	French
`fr-CA`	French Canadian
`it`	Italian

Locale Qualifier	Language Name
`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 (formerly Apptentive)

NuGet

Using the NuGet package manager is the easiest way to integrate. Follow the official Microsoft Visual Studio NuGet guide to add `Apptentive.iOS` package to your project.

Initialize Alchemer Mobile (Apptentive)

When your app starts, it will need to initialize the Alchemer Mobile SDK. You will need to provide your Alchemer Mobile (Apptentive) App Key and Alchemer Mobile (Apptentive) App Signature, available in the API & Development section under the Settings tab of the Alchemer Mobile Dashboard.

You will need to include “using” directive for `ApptentiveKit.iOS` namespace and create an instance of the `ApptentiveConfiguration` class with your Apptentive App Key and Apptentive App Signature. Then call the `Apptentive.Shared.Register()` method. We recommend registering Apptentive in your application delegate’s `FinishedLaunching()` method:

```
using ApptentiveKit.iOS;

[Register("AppDelegate")]
public class AppDelegate : UIApplicationDelegate
{
    public override bool FinishedLaunching(UIApplication application, NSDictionary launchOptions)
    {
        ...

        var configuration = new ApptentiveConfiguration("Your Apptentive Key", "Your Apptentive Signature");
        Apptentive.Shared.Register(configuration, null);

        return true;
    }
}
```

> Make sure you use the Alchemer Mobile (Apptentive) App Key and Alchemer Mobile (Apptentive) 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.

```
public partial class ViewController : UIViewController
{
    ...

    public override void ViewDidLoad()
    {
        base.ViewDidLoad();

        MessageCenterButton.TouchUpInside += delegate
        {
            Apptentive.Shared.PresentMessageCenter(this);
        };
    }
}
```

Unread Message Count Notification

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

```
public partial class ViewController : UIViewController
{
    public override void ViewDidLoad()
    {
        base.ViewDidLoad();

        ...

        NSNotificationCenter.DefaultCenter.AddObserver(Constants.ApptentiveMessageCenterUnreadCountChan
gedNotification, (NSNotification obj) =>
        {
            UnreadMessagesTextView.Text = "Unread messages: " + Apptentive.Shared.UnreadMessageCount;
        });
    }
}
```

Alternatively, the Apptentive class returns a `UIView` instance from the `UnreadMessageCountAccessoryView()` 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

- void SendAttachment(NSData fileData, string mimeType)

Hidden Image Messages

- void SendAttachment(UIImage image)

Hidden Text Messages

- void SendAttachment(string 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 engage an event when a view controller appears, for example in the view controller's `ViewDidLoad()` method:

```
public override void ViewDidLoad()
{
    base.ViewDidLoad();

    // 1. Engage the "viewed_list" event.
    Apptentive.Shared.Engage("viewed_list", this);

    // 2. Engage the "tapped_button" when the Button outlet is tapped
    Button.TouchUpInside += delegate {
        Apptentive.Shared.Engage("tapped_button", this);
    };

    // 3. Dismiss the view controller when the SaveButton is tapped
    SaveButton.TouchUpInside += delegate {
        this.DismissViewController(true, delegate
        {
            // Engage the event after the view has been dismissed
            Apptentive.Shared.Engage("saved_item", this.PresentingViewController);
        });

    }

}
```

Another good place to engage events is from the action methods in your view controllers, as shown in the second example above.

In the third example, 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.

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:

```
public void ViewControllerSelected(UITabBarController tabBarController, UIViewController viewController)
{
    if (tabBarController.ViewControllers[0] == viewController)
    {
        // Engage the "photos_tab_selected" event.
        Apptentive.Shared.Engage("photos_tab_selected", tabBarController);
    } else {
        // Engage the "favorites_tab_selected" event.
        Apptentive.Shared.Engage("favorites_tab_selected", tabBarController);
    }
```

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

```
if (!Context.Save(out error))
{
    Apptentive.Shared.Engage("core_data_save_failed", this);
}
```

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. 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.

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 Dialogs

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 `NSNotificationCenter`:

```
NSNotificationCenter.DefaultCenter.AddObserver(Constants.ApptentiveSurveySentNotification, (NSNotificatio
n obj) => {
    // Called when the user submits a survey
});
```

Notes

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

> See: How to Use Notes

Push Notifications

Alchemer Mobile can send push notifications to your app when you reply to your customers. Your replies are more likely to be seen by your customer when you do this. To set up push notifications, you will need to supply your push credentials on the Integrations page of your Alchemer Mobile dashboard, send us the ID that your push provider uses to identify the device, and call into our SDK when a user opens a push notification.

To use Alchemer Mobile push, you will need to add code to your application delegate, configure your app for push in the developer portal, and supply your push certificate and private key in your Alchemer Mobile dashboard.

Configuring Your Application Delegate for Push

Your app will have to register for remote notifications (we recommend registering for alert and sound notifications) as follows:

```
var pushSettings = UIUserNotificationSettings.GetSettingsForTypes(UIUserNotificationType.Alert | UIUserNoti
ficationType.Sound, new NSSet());

UIApplication.SharedApplication.RegisterUserNotificationSettings(pushSettings);
UIApplication.SharedApplication.RegisterForRemoteNotifications();
```

When the registration succeeds, your application delegate will have to pass the device token on to the Alchemer Mobile SDK:

```
 public override void RegisteredForRemoteNotifications(UIApplication application, NSData deviceToken)
{
    Apptentive.Shared.SetPushNotificationDeviceToken(deviceToken);
}
```

Your application delegate will also have to forward any push notifications that it receives to the Alchemer Mobile SDK:

```
 public override void DidReceiveRemoteNotification(UIApplication application, NSDictionary userInfo, Action<
UIBackgroundFetchResult> completionHandler)
{
    Apptentive.Shared.DidReceiveRemoteNotification(userInfo, this.Window.RootViewController, completionHa
ndler);
}
```

Your app will also have to forward any local notifications to the Alchemer Mobile SDK. There are three options for doing this.

The simplest option, if your app targets iOS 10 and later and does not use push or local notifications for purposes other than the Alchemer Mobile SDK, is to simply set the Alchemer Mobile (Apptentive) singleton as the current user notification center's delegate:

```
 UNUserNotificationCenter.Current.Delegate = Apptentive.Shared;
```

If you are targeting iOS 10 and later using Apple's UserNotifications framework, and your app uses local or push notifications for non-Apptentive reasons, you will need create an object that implements the `UNUserNotificationCenterDelegate` protocol with the following methods and set it as the current user notification center's delegate:

```
 public void DidReceiveNotificationResponse(UNUserNotificationCenter center, UNNotificationResponse respo
nse, Action completionHandler)
{
    var handledByApptentive = Apptentive.Shared.DidReceveUserNotification(response, completionHandler);
    if (!handledByApptentive) {
        // Handle the notification response
        completionHandler();
    }
}

public void WillPresentNotification(UNUserNotificationCenter center, UNNotification notification, Action compl
etionHandler)
{
    var handledByApptentive = Apptentive.Shared.WillPresentNotification(notification, completionHandler);
    if (!handledByApptentive) {
        // Decide how to present the notification
        completionHandler(UNNotificationPresentationOptions.Alert);
    }
}
```

In place of the `viewController`, you should determine which of your app's view controllers is currently being displayed and pass that in as the value for the `from` argument.

In place of `this.Window.RootViewController`, you should determine which of your app's view

controllers is currently being displayed and pass that in as the value for the from argument. You can pass nil for the view controller parameter and the SDK will create a new window to present Message Center in.

If your app supports both Alchemer Mobile notifications and notifications from another source, the Alchemer Mobile "DidReceive" methods above return a boolean value that indicates that the notification was intended for the Alchemer Mobile SDK. For methods that accept a completionHandler, it will be called by the Alchemer Mobile SDK only if the notification is intended for it, so it is your app's responsibility to call the completion handler in the event that the Apptentive method returns false.

Provisioning Your App For Push

In the Apple Developer Portal's Certificates, Identifiers and Profiles section, configure your app's App ID for push. The Alchemer Mobile push server only supports sending push notifications to the production environment, so you will not need to configure it for the development push environment.

Then create a new provisioning profile for your app. We recommend creating an ad-hoc provisioning profile for testing push in addition to the iOS App Store profile you will need to submit your app to the App Store. Download the provisioning profile(s) you created and then open them in Xcode.

Next, select your app project in Xcode's Project Navigator and select your app target from the list (or dropdown menu, if the list is collapsed). On the General tab, choose the provisioning profile you just created in the Signing section. Then switch to the Capabilities tab and turn on Push Notifications, and in the Background Modes section, select Remote Notifications.

Supplying Push Certificate and Private Key

Finally, you'll need to export the push certificate you created as part of the provisioning process and upload it to your Alchemer Mobile dashboard.

To do this, launch Keychain Access and choose the My Certificates category in the "login" keychain. You will see a certificate with a name like "Apple Production IOS Push Services: com.my.app.id" (where com.my.app.id is your app's bundle indentifier). Select it and choose File > Export Items.... You will need to encrypt the exported certificate and private key with a password of your choosing, and export in the PKCS-12 (.p12) format.

Then go to the Integrations page of your Alchemer Mobile dashboard in your browser, and expand the Alchemer Mobile (Apptentive) 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

Because the Alchemer Mobile push service works only with the production environment, you will have to take a few extra steps to be able to test it.

You will need to compile using an ad-hoc provisioning profile for signing (configured in the Project Options dialog), and you will need to use the Release configuration when running.

At this point you should be able to run your app, send a message in Message Center and close your app, reply in your Alchemer Mobile dashboard, and receive a push notification.

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 either the device, or the person using the app. This is useful for sending user IDs and other information that helps you 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`.

```
 Apptentive.Shared.AddCustomPersonData("Seattle", "city")
Apptentive.Shared.AddCustomPersonData(500, "points")
Apptentive.Shared.AddCustomPersonData(true, "is_premium")

Apptentive.Shared.AddCustomDeviceData("test@apptentive.com", "primary_account")
Apptentive.Shared.AddCustomDeviceData(5, "user_count")
Apptentive.Shared.AddCustomDeviceData(false, "full_version")
```

Other

Logging

The Alchemer Mobile SDK has six logging levels: Verbose, Debug, Info, Warn, Error, and Crit. You can set the `LogLevel` on the `ApptentiveConfiguration` object that you pass to the `register` method to override the log level.

Potentially-private logging information will be hidden automatically when a debugger is not attached to the app. You can override this behavior by setting `ShouldSanitizeLogMessages` on the `ApptentiveConfiguration` object to `true` .

Migrating to version 6

Version 6 introduces to important changes.

First you will have to update your `using` statements to use `ApptentiveKit.iOS` in place of `ApptentiveSDK.iOS` . This better matches the name of the native iOS framework that it wraps.

Second, the `register` method is now an instance method on the `Shared` singleton to better match iOS platform conventions as well as the native ApptentiveKit framework. The new `register` method also takes a second parameter, which is a (nullable) callback that is called with the results of the register operation.

Related Articles