ZBi for MARKETERS

SDK Documentation

ZUMOBI

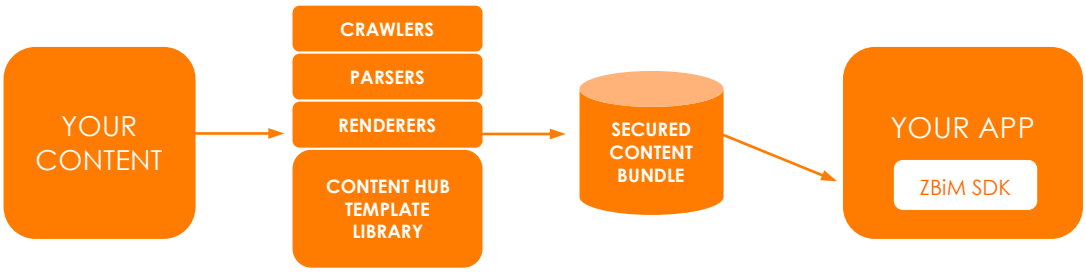
TABLE OF CONTENTS

OVERVIEW

This document provides an overview of ZBi for Marketers (ZBiM) SDK and describes the way it can be integrated into and interacted with by a client application.

The ZBiM SDK provides the ability to deliver tailored content inside a native application, where the content is managed directly by the marketing team responsible for the application. The goal is to bridge the gap between content creation and content delivery in an intuitive, transparent, and scalable way while allowing for personalization based on analyzing users' behaviors without involving the application development team or IT beyond the initial SDK integration.

Below is a high level diagram that describes the path that the content travels through the ZBiM Platform from the content source to the SDK integrated inside your native application:



In the above diagram, the Zumobi Publishing Engine fetches content (this content can be from your CMS, a Zumobi content partner, a social media source, or the Zumobi cloud-based CMS), processes it into a secure form by applying pre-crafted Content Hub templates, and delivers the rendered content bundle to the SDK over a trusted pipeline for creating the custom-crafted, walled-garden experience within your native application.

REQUIREMENTS

ZBiM SDK requires:

- Xcode 5 or later
- iOS 7 or later.

PROJECT SETUP

This document assumes that you have already created an iOS project in Xcode.

For more information on downloading, installing, and getting started with iOS development see the official iOS developer documentation at: https://developer.apple.com/devcenter/ios/.

The following steps are used to reference the ZBiM SDK project in an application target:

1. Copy the *zbim* directory from the *zbim.zip* archive into your own project directory.
2. With the desired destination selected in Xcode's *Project Navigator*, select *Add files to <project>...* from either the *Project Navigator* context menu or Xcode's *File* menu.
3. Browse and select the *zbim* directory.
4. With your project selected in *Project* Navigator, select the appropriate target, navigate to the *Build Phases* section and expand the *Link Binary With Libraries*. Add the *libzbim.a* static library by pressing on the "+" sign at the bottom of the list and locating *libzbim.a* under the *zbim* folder.
5. Add the following list of frameworks to the *Link Binary With Libraries* list under the *Build Phase tab:*
    - Accounts
    - AdSupport
    - EventKit
    - EventKitUI
    - MediaPlayer
    - MessageUI
    - PassKit
    - Social
    - StoreKit
    - SystemConfiguration
    - Twitter
6. Add the following dynamic library to the Link Binary With Libraries list:
    - libsqlite3.dylib
    - libz.dylib

7. While still in the Build Phases section, expand the *Copy Bundle Resources* group and add *zbimResources.bundle* by clicking on the "+" sign.
8. Next navigate to the *Build Settings* tab and add the *zbim/include* directory to the *Header Search Paths* setting.
9. With the Build Settings tab still selected, add *-ObjC* to the *Other Linker Flags*.

QUICK START

Here's the minimum set of steps required to get a Content Hub up and running:

1. Create a ZBiM configuration file named zbimconfig.plist and make sure it is included as part of the application's bundle (see *Build Phases*, *Copy Bundle Resources*). The file must contain the following entries:
    a. dbServiceUrl – a Zumobi-provided URL to content download service
    b. appID – a Zumobi-provided ID needed for metrics reporting
    c. metricsReportingQueueId – a Zumobi-provided ID also needed for metrics reporting
2. Add the Zumobi-provided pubkey.der file to the Xcode project and make sure it is included as part of the application's bundle (see *Build Phases*, *Copy Bundle Resources*).
3. Initialize the ZBiM SDK. Inside your AppDelegate class add the following:

```
#import "ZBiM.h"

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary
*)launchOptions
{
    [ZBiM start];
    // ...
    return YES;
}
```

4. Inside the view controller for the view containing the Content Hub's entry point, add:

```
#import "ZBiM.h"

- (void)viewDidLoad
{
    [super viewDidLoad];

    if (![ZBiM activeUser])
    {
        NSError *error = nil;
        NSString *userID = [ZBiM generateDefaultUserId];
        if (![ZBiM createUser:userID withTags:nil error:&error])
        {
            NSLog(@"Failed creating user: %@", error);
        }
        else
        {
            if (![ZBiM setActiveUser:userID error:&error])
            {
                NSLog(@"Failed setting active user: %@", error);
            }
        }
    }
}
```

5. Create an entry point for the Content Hub, e.g. a button, whose touch-up-inside action handler is defined as follows:

```
- (IBAction)showContentHubPressed:(id)sender
{
    [ZBiM presentHubWithTags:nil completion:^(BOOL success, NSError *error) {
        if (!success)
        {
            NSLog(@"Failed presenting content hub. Error: %@", error);
        }
    }];
}
```

INITIALIZATION

Include the ZBiM.h header in the source where you will be using the ZBiM SDK with the following directive: #import "ZBiM.h".

To use the ZBiM SDK, it must first be initialized by calling:

```
+ (void) start;
```

This is done once per application session and is recommended that it is done as early as possible, e.g. inside:

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary
*)launchOptions
{
    [ZBiM start];
    // ...
    return YES;
}
```

The first time the SDK is initialized, the client application must create a user ID and set it as the active user. This user ID is used for tracking the user's activity while interacting with the Content Hub experience for metrics reporting and content tailoring purposes. The user ID is used in an anonymous fashion and the SDK has a built-in mechanism (described below) to generate a new user ID for this purpose if you do not have or cannot use an existing user ID.

If the client application allows multiple users to be logged in over time, a different user ID can be registered with the SDK and associated with each application user, enabling the SDK to preserve tailoring data in case a previously registered user comes back. An important thing to note is that while there can be multiple user IDs registered with the SDK, only one of them can be active at any point in time.

To create a new user, call:

```
+ (BOOL) createUser:(NSString *)userId withTags:(NSArray *)tags error:(NSError * __autoreleasing *)error;
```

If the application already has the concept of a user ID, e.g. the application requires the user to login to access services provided by the application, the application can pass that value (username or assigned unique identifier) as the ZBiM user ID.

It is important to note that it is the application's responsibility to ensure that no personally identifiable information is passed to ZBiM SDK. The username provided will be used as-is and no attempt at obfuscating the value will be made by the ZBiM SDK.

Alternatively the application can call the following method provided by the ZBiM SDK to generate a new user ID:

```
+ (NSString *) generateDefaultUserId;
```

Once the user ID has been created, the application must call the following method in order to instruct the SDK to start using the newly create user ID:

```
+ (BOOL) setActiveUser:(NSString *)userId error:(NSError * __autoreleasing *)error;
```

The client application can check if there is an active user already set by calling:

```
+ (NSString *) activeUser;
```

To summarize, the pattern looks as follows:

```
if (![ZBiM activeUser])
{
    NSError *error = nil;
    NSString *userID = [ZBiM generateDefaultUserId];
    if (![ZBiM createUser:userID withTags:nil error:&error])
    {
        NSLog(@"Failed creating user: %@", error);
    }
    else
    {
        if (![ZBiM setActiveUser:userID error:&error])
        {
            NSLog(@"Failed setting active user: %@", error);
        }
    }
}
```

USER TAGS & CONTENT TAILORING

When creating a new user, the application has the option to pass a set of tags that sets up the user's profile when the user launches the application for the first time. This is to provide a hint to the ZBiM SDK on how it should initially customize the experience for that particular user. Such tags can initially be derived from user data that is already available to the marketer (e.g. "sports fan" or "high income"), but may change over time as the user interacts with the Content Hub experience. Tags are central to delivering tailored content to the user, however, providing these to the SDK is optional and can be skipped, in which case the ZBiM SDK will not start with a customized experience for the user.

The application also has the option to query ZBiM SDK for the set of tags accumulated until that point by calling:

```
+ (NSArray *)tagsForUser:(NSString *)userId error:(NSError * __autoreleasing *)error;
```

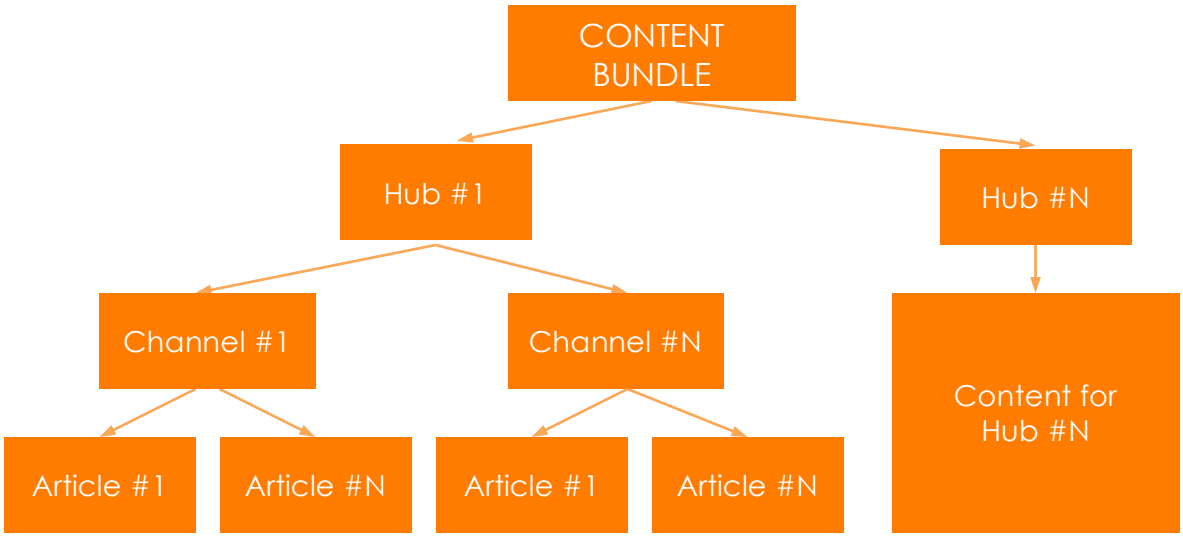
An important requirement for ZBiM SDK to function properly is that client code and content source must use the exact same set of tags. If there is a divergence, then the ZBiM SDK will likely not select the most appropriately tailored content and in certain case may not show any content at all.

CONTENT STRUCTURE

When content feeds are presented in-app to a user via the ZBiM SDK, we call this area of the application a Content Hub. In some cases, a brand might have multiple Content Hubs, which are expressed in one or more individual applications.

Content is organized logically in a hierarchy. The Content Hub loads data from a content source. A content source can have one or more hubs. A hub can have one or more channels and each channel can have one or more articles.

At each level of the hierarchy there is a set of tags associated with the corresponding item, e.g. hub, channel or article. When the time comes for the SDK to decide which hub to show or how to priority-order articles inside a channel these tags are compared against the set of tags associated with the current active user.



CONTENT HUB UI & NAVIGATION MODEL

Content is presented to the user inside a Content Hub, which can be in the form of a modal view, taking over the entire screen, or nested inside a parent view, provided by the client application. It is up to the application to decide which mode is more appropriate in the context of what it is trying to achieve.

To display the Content Hub as a modal view, call:

```
+ (void) presentHub:(void (^)(BOOL success, NSError *error))completionCallback;
```

By default the ZBiM SDK will find a presenting view controller on behalf of the host application and use it. If you need to override the default behavior, call:

```
+ (void) presentHubWithPresentingViewController:(UIViewController *)presentingViewController
                                      completion:(void (^)(BOOL success, NSError *error))completionCallback;
```

To present the Content Hub nested in a parent view, call:

```
+ (id<ZBiMContentHubDelegate>) presentHubWithParentView:(UIView *)parentView
                                   parentViewController:(UIViewController *)parentViewController
                                             completion:(void (^)(BOOL success, NSError *error))completionCallback;
```

By default, when presented, the Content Hub will load the hub page that best matches the set of tags associated with the current active user. If tags are not available, then the first available hub page will be selected, based on internal ordering specified in the content bundle.

There are two variations of the above three methods, which allow changing the hub page selection logic. These allow passing a specific URI, which can point to a hub, channel or article page or a set of tags, which will override the tags associated with current active user, which are used to select the most relevant hub page to load.

To display the Content Hub as a modal view using a tags override, call:

```
+ (void) presentHubWithTags:(NSArray *)tags
                 completion:(void (^)(BOOL success, NSError *error))completionCallback;
```

To present the Content Hub as a modal view using a tags override and passing a presenting view controller, call:

```
+ (void) presentHubWithTags:(NSArray *)tags
   presentingViewController:(UIViewController *)presentingViewController
                 completion:(void (^)(BOOL success, NSError *error))completionCallback;
```

To present the Content Hub nested in a parent view, using a tags override, call:

```
+ (id<ZBiMContentHubDelegate>) presentHubWithTags:(NSArray *)tags
                                       parentView:(UIView *)parentView
                               parentViewController:(UIViewController *)parentViewController
                                       completion:(void (^)(BOOL success, NSError *error))completionCallback;
```

To present the Content Hub as a modal view using an URI override, call:

```
+ (void) presentHubWithUri:(NSString *)uri
                 completion:(void (^)(BOOL success, NSError *error))completionCallback;
```

To present the Content Hub as a modal view using an URI override and passing a presenting view controller, call:

```
+ (void) presentHubWithUri:(NSString *)uri
  presentingViewController:(UIViewController *)presentingViewController
                 completion:(void (^)(BOOL success, NSError *error))completionCallback;
```

To present the Content Hub nested in a parent view, using an URI override, call:

```
+ (id<ZBiMContentHubDelegate>) presentHubWithUri:(NSString *)uri
                                       parentView:(UIView *)parentView
                               parentViewController:(UIViewController *)parentViewController
                                       completion:(void (^)(BOOL success, NSError *error))completionCallback;
```

In the case where a previously presented Content Hub has been dismissed and a strong reference to it has been kept around, call the presentExistingContentHub:completion: method. That way, users can continue interacting with the Content Hub where they left off:

```
+ (void) presentExistingContentHub:(id<ZBiMContentHubDelegate>)existingContentHub
                 completion:(void (^)(BOOL success, NSError *error))completionCallback;
```

Notice that only the method calls presenting the hub as a nested view, return a reference to an object implementing the ZBiMContentHubDelegate protocol. The protocol defines the following methods for interacting with the Content Hub:

```
- (void) goBack;
- (BOOL) canGoBack;
- (void) goForward;
- (BOOL) canGoForward;
- (BOOL) dismiss:(NSError * __autoreleasing *)error;
- (void) setScrollDelegate:(id<UIScrollViewDelegate>)delegate;
- (void) setBackgroundColor:(UIColor *)color;
```

The reason for the first method not returning the same type of object is that such object is not needed in the context of a full-screen Content Hub since all of the above-methods are handled internally by the Content Hub itself.

ZBiM SDK keeps all navigation internal to the Content Hub except for the following cases:

1. Videos are played full-screen in a separate native modal view.
2. Clicking on external links will prompt the user to leave the application and be navigated to the default browser.
3. Clicking on a link which implements a custom scheme, e.g. zbim_sample_app://URI will provide the capability of Content Hub to talk back to the application, notifying the application of an action that it needs to handle.

Only one Content Hub can be shown by the SDK at any point in time. Multiple side-by-side hubs or stack of hubs displayed on top of each other are not supported.

The default Content Hub view has a predefined user interface (close, back buttons, ZBiM logo). In case the application wants to customize the Content Hub UI so it fits more seamlessly with the rest of the UI, the application should provide a parent view and corresponding parent view controller and the Content Hub will insert itself, taking 100% of the provided view and not showing any buttons. It is the application's responsibility to show the corresponding navigation buttons and connect them to the methods exposed by the ZBiMContentHubDelegate protocol.

Please note that when using the above-mentioned mode (where the Content Hub is hosted inside an application-provided parent view), it is not enough to simply dismiss the parent view's controller and rely on it doing all the necessary cleanup for you. It is imperative that application code (typically part of the parent view's controller) must call the ZBiMContentHubDelegate's dismiss method, i.e.:

```
- (BOOL) dismiss:(NSError * __autoreleasing *)error;
```

on the object implementing the ZBiMContentHubDelegate protocol. If the Content Hub is to be permanently dismissed, the application must not keep any strong references to the object implementing the ZBiMContentHubDelegate, so it can be properly accounted for and disposed of. If on the other hand the application intends to keep the Content Hub around and present it again in the future, then application code must still call the dismiss method, but keep a strong reference around. When the application is ready to have the Content Hub presented again, it should call:

```
+ (void) presentExistingContentHub:(id<ZBiMContentHubDelegate>)existingContentHub
                        completion:(void (^)(BOOL success, NSError *error))completionCallback;
```

In certain cases an embedded Content Hub may need to communicate back to its parent view controller, informing it that the Content Hub needs to have a certain action performed on its behalf. Currently the only such supported action is closing the Content Hub and for the parent view controller to be able to respond, it needs to implement the ZBiMContentHubContainerDelegate protocol. The reasoning for this design is that the parent view controller knows best how to perform certain actions and these should be delegated to it.

Let's take as an example dismissing the Content Hub from within the Content Hub's embedded view controller. To do this properly, the Content Hub needs to figure out how the container view controller was presented, e.g. if it was presented modally, pushed on a navigation controller or added as a child of another view controller. The Content Hub will also need to decide if it needs to dismiss only itself or if the container view controller should be dismissed as well. The Content Hub also needs to account for how dismissing the container view controller should be animated, if at all. Finally, the Content Hub will have to know whether it will be reused in the future, in which the strong reference to it must be preserved or if it is no longer needed and should thus be discarded. Clearly these are all decisions better suited for the container view controller to take care of.

Progress updates, e.g. indicating that new content is being downloaded, will be displayed inside the Content Hub UI. If content cannot be presented for any reason, a message will be displayed inside the Content Hub UI, informing users that an error occurred. The Content Hub has generic views for all status-related UI. The host application can, however, register itself as a provider of such views and override the ones provided by default. For more details please see Customizations section.

SECURITY

Zumobi realizes the sensitive nature of serving content within the framework of an existing application and the various degrees to which a client application can be affected by a malicious or otherwise inappropriate content being served.

We take security very seriously and have included features that give an application's administrators direct control over the usage of the SDK with the application and the ability for the application itself to reject content that has been altered by an unauthorized party.

A basic premise of the core architecture of ZBiM is that web technologies expose a large number of possible vectors for attack, interception, and content tampering. From social engineering to technical issues that reveal vulnerabilities of the underlying application code, there are numerous ways a client application can be impacted. Rather than attempting to address every attack vector separately, we took a more robust approach by creating a secured walled-garden content experience that is delivered by a system that secures each step of the content's journey until it reaches the native application.

All metadata needed to verify the integrity of the content is exchanged over a secured network connection (TLS). The metadata is then used to verify that the content itself has not been tampered with. For these additional checks to work the application bundle needs to contain a Zumobi-provided file that contains a public key specific to each application. The name of the file is pubkey.der and it must be carried as a resource by your application bundle.

Once content has made it to the device and is ready to be consumed, ZBiM SDK will ensure that the major content types, i.e. hub, channel or article can only be loaded from the content bundle, whose integrity has already been verified. By default, the ZBiM SDK will allow external resources to still be accessed, e.g. fetching remote data, playing a YouTube video, etc. For applications that do not require access to external resources and/or have heightened security requirements, ZBiM SDK provides the option to deny access to any resource that's not part of the local content bundle. Thus, in the unlikely case where malicious code manages to sneak into a brand's Content Hub, any attempts to communicate with systems outside of the hub will be denied. A goal of this architecture is to hold the branded content portion of the application to the same standard of trustworthy computing as other mission-critical, secured features within the existing application.

Deciding whether access to external resources is permitted or not is based on the value of the ZBiM's contentSource property. By default, its value is ZBiMContentSourceExternalAllowed. By changing it to ZBiMContentSourceLocalOnly the application can prevent the Content Hub from accessing any resources outside of the local content bundle's scope:

```
[ZBiM setContentSource:ZBiMContentSourceLocalOnly];
```

In addition to the above, ZBiM also provides the option to remotely shut down your specific implementation of the Content Hub, which prevents the hub from appearing within your application. This gives the administrator the immediate ability to remove the Content Hubs from your application in the event that some inappropriate or unauthorized content was discovered.

Finally, for applications that want elevated security measures, but still need an occasional exception in terms of what is allowed (or not) to go through on the network, there is a hybrid option where contentSource is kept at ZBiMContentSourceLocalOnly, but individual URLs or whole domain names can be whitelisted. This allows the host application to inform the ZBiM SDK to not make these subject to the same security restrictions.
To whitelist a specific URL, the application must call ZBiM's whitelistURL method, e.g.:

```
[ZBiM whitelistURL:[NSURL URLWithString:@"http://www.zumobi.com/about"]];
```

Please note that when a URL is checked against the list of whiltelisted URLs, the entire URL, i.e. including any parameters, anchors, etc. will be used verbatim as provided to the method call above.

If the previous method is too restrictive, then the application has the option to whitelist an entire domain by calling ZBiM's whitelistDomainName method. Here's an exmaple:

```
[ZBiM whitelistDomainName:@"www.zumobi.com"]];
```

CUSTOM URL SCHEMES SUPPORT

The security measures descibed in the previous section can impact ZBiM SDK's ability to support custom URL schemes. The SDK has the ability to pass URLs using a custom scheme to the host application for the purpose of performing actions not supported directly by the SDK itself. For example, the Content Hub may display an item available for purchase, but only the application can perform the checkout action. If the user taps on the purchase button within the Content Hub, the ZBiM SDK will generate a custom URL request for the host application to handle the checkout functionality.

By default the ZBiM SDK will read all custom URL schemes (from the app-info.plist file) that the app has registered to handle and add them to a list of whitelisted URL schemes that are exempt from the security restrictions described in the previous section. The application can add more URL schemes that should be treated the same way by calling:

```
+ (void) registerCustomURLScheme:(NSString *)customURLScheme;
```

CONFIGURATION

Configuration data must go into zbimconfig.plist file, which is carried as a resource by the application bundle. The supported set of configuration entries is:

1. dbServiceUrl – a Zumobi-provided URL to content download service
2. appID – a Zumobi-provided ID needed for metrics reporting
3. metricsReportingQueueId – a Zumobi-provided ID also needed for metrics reporting

CUSTOMIZATIONS

There are several main aspects in which the ZBiM SDK's look and behavior can be customized. These are:

1. The Content Hub's default navigation chrome can be omitted entirely in favor of an application-provided one. This is achieved by nesting the Content Hub into a parent view and was already discussed in the Content Hub UI & Navigation Model section.

2. The Content Hub UI supports two different color schemes - dark (default) and light - allowing the application to choose which one fits better with the existing look and feel. The color schemes selection affects the Content Hub's navigation chrome as well as the built-in progress reporting and error messaging. Supported values for color schemes are ZBiMColorSchemeDark (default) and ZBiMColorSchemeLight. Application can query the current color scheme by calling:

   ```
   + (ZBiMColorScheme) colorScheme;
   ```

   or it can set the active color scheme by calling:

   ```
   + (void) setColorScheme:(ZBiMColorScheme)colorScheme;
   ```

   The Content Hub UI can be further customized by calling ZBiMContentHubDelegate's setBackgroundColor: method, which allows the application to customize the background color for the Content Hub's embedded UIWebView:

   ```
   - (void) setBackgroundColor:(UIColor *)color;
   ```

3. The client application can register itself as a provider of status reporting views, which the Content Hub uses to communicate to the user one of three things - it is checking for new content, downloading new content or it has encountered an error. To do that, the host application needs a class implementing the ZBiMContentHubStatusUIDelegate protocol:

   ```
   @protocol ZBiMContentHubStatusUIDelegate <NSObject>
   - (UIView *) getErrorView:(NSString *)optionalMessage;
   - (UIView *) getCheckingForContentView:(NSString *)optionalMessage;
   - (UIView *) getDownloadProgressView:(NSString *)optionalMessage
                        percentCompleted:(CGFloat)percentCompleted;
   @end
   ```

It then needs to register an instance of that class with the ZBiM SDK by calling:

```
+ (void)
setCustomStatusUIDelegate:(id<ZBiMContentHubStatusUIDelegate>)delegate;
```

When the Content Hub needs to present any of the three status reporting views it will call into the above mentioned object using one of the corresponding methods and passing as parameter data the application can use to construct/update the view. For example, when reporting download progress, the SDK will call with a default message of "Waiting for new content to be downloaded" and provide a percent-completed value, between 0.0 and 1.0. The application can then choose to use the data passed as parameters to update a label and a progress bar. Alternatively it can ignore it and show whatever makes most sense in the context of what it is doing.

4. The ZBiM SDK's logging severity and verbosity levels can be adjusted as well as the actual logging method, e.g. write to debug console or pop up a native dialog view. For more details see the Logging section later in this document.

5. The application can chose between two different content download modes – non-blocking and blocking.
    a. Non-blocking sync mode leaves it to the SDK to figure out when and how often to check for and download new content. When the application requests that the SDK present the Content Hub, the SDK will use whatever content is available at the time of the request. Once presented the Content Hub will not be affected by subsequently downloaded content unless it is dismissed and presented again. The only case when the Content Hub will show a download-in-progress UI is when there's no content that can be presented.
    b. Blocking sync mode is more deterministic in terms of when content download takes place and what gets presented to the user. In this mode the Content Hub will check for new content every time it is being presented and if new content is available, the Content Hub will block, showing the user a download-in-progress UI, until the download completes.

Auto-sync is the default sync mode. To change it call:

```
+ (void) setSyncMode:(ZBiMSyncMode)syncMode;
```

To find out what is the current sync mode, use:

```
+ (ZBiMSyncMode) syncMode;
```

6. While the use of non-blocking download mode allows the user to see content pretty much instantaneously, there is still the case where no content has been previously downloaded, in which case the user will still be presented with the download-in-progress UI. The application has the option to avoid this by including a file named contentDB.sqlite3 as part of the application's main bundle. If present, content from the file will be loaded and presented to the user immediately, while new content is downloaded asynchronously. For that to work, the client application must use the default download mode of non-blocking.

7. The SDK allows the application to determine what constitutes a meaningful advertiser ID and pass the value back to the SDK to be used for reporting purposes. To take advantage of that customization option, the application must implement the ZBiMAdvertiserIdDelegate protocol and set itself as the corresponding advertiser ID delegate.

REACTING TO ZBIM SDK STATE CHANGES

ZBiM SDK provides limited access to internal state-related information that host application can use to adjust UI to improve the user experience. This section describes publicly accessible state information and provides examples of how it can be used.

The following four methods allow access to a subset of ZBiM SDK internal states.

1. Indicates whether content source related metadata or content source itself is being currently downloaded:

```
+ (BOOL) isDownloadingContent;
```

2. Indicates if Content Hub is currently being displayed. It provides no information regarding what the hub is showing, i.e. actual content, progress indicator or error status:

```
+ (BOOL) isDisplayingContentHub;
```

3. Indicates whether SDK is ready to show content. Implies that a valid content has been already downloaded and validated and is ready to be presented.

```
+ (BOOL) isReady;
```

4. Indicates whether SDK has been shutdown remotely (typically as a security precaution):

```
+ (BOOL) isDisabled;
```

Please note that more than one of the above can return true simultaneously, e.g. content might be downloading while the Content Hub is being presented, but there's previously downloaded valid content, so the ZBiM SDK is in a "ready" state.

For each of the above-mentioned states there is a notification that client code can subscribe to if it needs to know when the corresponding state changes. The notifications' names are:

1. ZBiMDownloadingContentStateChanged
2. ZBiMDisplayingContentHubStateChanged
3. ZBiMReadyStateChanged
4. ZBiMDisabledStateChanged

One example of how the application can take advantage of the above is by showing or hiding the entry point (e.g. a button) to the Content Hub. If the ZBiM SDK has been remotely disabled, we know the application will not download and display a Content Hub for the remainder of the application session. In such a scenario, the entry point can be hidden altogether until any security concerns have been resolved to provide the user with a more seamless app experience until the Content Hub is again accessible.

FACILITATING AN ADAPTIVE APPLICATION UI

In addition to state-related information, the ZBiM SDK can notify host application when certain events of interest take place. This allows the application to adapt its UI in order to provide a more seamless ZBiM SDK integration and is achieved via the following two notifications:

1. ZBiMDBDownloadProgressChanged
2. ZBiMContentTypeChanged

ZBiM SDK also provides access to the Content Hub's raw scroll events.

If the application needs to visualize the content download progress, it can subscribe to the ZBiMDBDownloadProgressChanged notification. The NSNotification object's userInfo property bag contains a key named "progress", which is a float value between 0.0 and 100.0. The event will be fired every time when there's a notable change in download progress.

ZBiM SDK also allows an application-provided class to set itself as a UIScrollViewDelegate to the UIWebView that renders the Content Hub. This is targeted primarily at the nested mode, where the Content Hub is presented inside a parent view. By getting access to the raw scrolling events, the parent view can implement behaviors such as hiding and showing the navigation chrome depending on the scroll direction. A sample implementation can be found as part of the ZBiM Sample Application's source code.

To set a class as a UIScrollViewDelegate, the application needs to call the following ZBiMContentHubDelegate's method:

```
- (void) setScrollDelegate:(id<UIScrollViewDelegate>)delegate;
```

Finally, the application can subscribe for the ZBiMContentTypeChanged notification, which gets triggered every time the Content Hub navigates to a new piece of content. The notification's payload includes three pieces of information:

1. Content type indicates whether a hub, channel or article is being loaded and is represented by the ZBiMResourceTypeHub, ZBiMResourceTypeChannel and ZBiMResourceTypeArticle enum values.
2. Content URL provides a way to identify and locate the specific piece of content.
3. Content title is a UI-friendly string that can be used elsewhere in the host application to describe the content being loaded.

By monitoring for changes to the currently loaded content item, the application can implement, for example, conditional logic to disable the hiding and showing animation for the navigation chrome when user is reading an article and re-enable it when user navigates back to channel or hub views. It can also extract the content's title and include it as part of its UI. A sample implementation of the above-mentioned scenario can be found in the ZBiM Sample App's source code.

LOGGING

Logging is designed to be pretty flexible, allowing the client application to decide what gets logged and how.

The client application can set the logging delegate by calling:

```
+ (void)setLoggingDelegate:(id<ZBiMLoggingDelegate>)loggingDelegate;
```

The client application also has the option to set the logging severity and verbosity levels via:

```
+ (void) setSeverityLevel:(ZBiMSeverityLevel)severityLevel;
+ (void) setVerbosityLevel:(ZBiMVerbosityLevel)verbosityLevel;
```

And to query the currently set values via:

```
+ (ZBiMSeverityLevel) severityLevel;
+ (ZBiMVerbosityLevel) verbosityLevel;
```

The values for logging severity are:

ZBiMLogSeverityNone
ZBiMLogSeverityCriticalError
ZBiMLogSeverityError
ZBiMLogSeverityWarning
ZBiMLogSeverityInfo

The values for logging verbosity are:

ZBiMLogVerbosityNone
ZBiMLogVerbosityDebug
ZBiMLogVerbosityInfo

The severity and verbosity levels determine what information will be logged and what ignored even before the SDK checks if a logging delegate has been set. By default severity is set to ZBiMLogSeverityError and verbosity to ZBiMLogVerbosityDebug. If either the severity or verbosity parameter value is lower than the currently set levels the call will be ignored. The stacking of the different values is per the ordered lists above. Setting either the severity or verbosity to none (e.g. ZBiMLogSeverityNone or ZBiMLogVerbosityNone) will have the effect of turning logging off.

If the application code wants full access over everything that is to be logged, then it can set severity and verbosity to the least restrictive values, i.e. ZBiMLogSeverityInfo and ZBiMLogVerbosityAlert correspondingly, and do all filtering inside the logging delegate object.

METRICS

The ZBiM SDK has an onboard metrics collection system that enables marketers to measure the performance of the various pieces of content and the interaction history of the user against the Content Hub for segmentation purposes. The SDK will periodically upload collected metrics to a cloud-based reporting system over a secured connection for marketing analysis. This secured, on-device metrics system is designed to work over mobile networks and is resilient against network interruptions or if the user puts his/her device into "airplane" mode.

For metrics collection to work properly, the client application must provide the reporting queue ID in the zbimconfig.plist file, whose value is to be assigned by Zumobi.

One metrics-related attribute that deserves special attention is the advertiser ID. As part of metrics reporting the ZBiM SDK collects the advertiser ID based on the user's OS preferences which can later be used for remarketing campaigns. For example, ZBiM SDK collects the current IDFA value, but only if user has not limited ad tracking (via Settings->Privacy->Advertising->Limit Ad Tracking). If Limit Ad

Tracking is turned on, then ZBiM SDK will not use the advertiser ID at all. This default behavior supported by the SDK can be overridden by the host application if desired. The application can simply provide its own behavior by implementing the ZBiMAdvertiserIdDelegate protocol and setting itself as the advertiser ID delegate by calling:

```
+ (void) setAdvertiserIdDelegate:(id<ZBiMAdvertiserIdDelegate>)advertiserIdDelegate;
```

LOCAL NOTIFICATIONS

ZBiM SDK supports scheduling of local notifications on behalf of the client application. Metadata describing when a local notification is to be scheduled, as well as what its message and action should be are provided via the ZBiM Portal.

There are only a couple of things an application needs to do in order to enable local notifications support:

1. If the application supports iOS 8 and later, it must register to receive local notifications. From Apple's documentation:

> "In iOS 8 and later, apps that use either local or remote notifications must register the types of notifications they intend to deliver."

Here is an example (from ZBiMAppDelegate.m) of how the ZBiM Sample Application implements this requirement:

```
#if __IPHONE_OS_VERSION_MAX_ALLOWED >= __IPHONE_8_0
   // In iOS 8 and later, the application must explicitly
   // register for notifications.
   if ([application respondsToSelector:@selector(registerUserNotificationSettings:)])
   {
      [application registerUserNotificationSettings:[UIUserNotificationSettings
settingsForTypes:UIUserNotificationTypeAlert categories:nil]];
   }
#endif
```

2. The native application also needs to implement the standard methods for handling local notifications, e.g.:

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary
*)launchOptions;

- (void)application:(UIApplication *)application didReceiveLocalNotification:(UILocalNotification
*)notification;
```

Once a local notification has been received, the application can pass it to ZBiM SDK for handling by calling one of the following methods:

```
+ (BOOL) handleLocalNotification:(UILocalNotification *)notification;
```

```
+ (BOOL) handleLocalNotification:(UILocalNotification *)notification showAlert:(BOOL)showAlert;
```

If the local notification was not scheduled by ZBiM SDK, the above-mentioned methods will return NO and let the applciation handle the notifications as it deems appropriate.

The first of the local notification handling methods is just a convenience wrapper for the second, calling it with showAlert set to YES. The case where handleLocalNotifications should be called with showAlert set to NO is if the user has already expressed interest in actioning on the local notification, e.g. from the notification bar. This tells ZBiM SDK that there is no need to show a prompt, asking the user if they want to action on the notification, but it should go straight to executing the action.

IN CONCLUSION

We hope that you've found the ZBiM SDK to provide an intuitive way to illuminate your array of branded content into your existing branded application. Should you need assistance with implementation, have platform feedback, or great ideas about new features that you and your team would find useful, we're here to help. Please feel free to contact us at zbim-support@zumobi.com.

Thanks for spending this time diving-in and implementing our SDK.