

Mobile Sample Application and SDK Reference Guide

**Based on Brandify Mobile SDK**

Last updated: January 28th, 2019

Table of Contents

[1 Version History 3](#_Toc491085945)

[2 Introduction 4](#_Toc491085946)

[3 SDK Core Functionality 4](#_Toc491085947)

[3.1 Configuration 4](#_Toc491085948)

[3.2 Customization 6](#_Toc491085949)

[4 Getting Started in iOS and Android 8](#_Toc491085950)

[4.1 Include the Brandify Mobile SDK 9](#_Toc491085951)

[4.1.1 iOS 9](#_Toc491085952)

[4.1.2 Android 10](#_Toc491085953)

[4.2 Initialize the Brandify Mobile SDK 11](#_Toc491085954)

[4.2.1 iOS 11](#_Toc491085955)

[4.2.2 Android 11](#_Toc491085956)

[4.3 Building the Distributed iOS Application Binary 13](#_Toc491085957)

[4.4 Configure the Brandify Mobile SDK Inside your Mobile Application 14](#_Toc491085958)

[4.5 The Brandify Sample Application 15](#_Toc491085959)

[5 Analyzing the iOS Sample Application 16](#_Toc491085960)

[5.1 Locator Searches 17](#_Toc491085961)

[5.2 Integrate with the iOS Map Kit 19](#_Toc491085962)

[5.3 Locator Searches using The Brandify Mobile SDK and iOS MapKit Integration 20](#_Toc491085963)

[5.4 Custom Callout 21](#_Toc491085964)

[5.5 Location Detail View 23](#_Toc491085965)

[5.6 Presenting Locator Search Results in a List View 24](#_Toc491085966)

[5.7 Filtering Locator Searches 25](#_Toc491085967)

[5.8 Retrieving Custom Properties 26](#_Toc491085968)

[5.9 Getlist Searches 27](#_Toc491085969)

[5.10 Accessing Configuration Parameters 28](#_Toc491085970)

[6 Analyzing the Android Sample Application 30](#_Toc491085971)

[6.1 Locator Searches 31](#_Toc491085972)

[6.2 Integrating with the Google Maps API 32](#_Toc491085973)

[6.3 Custom Callout 33](#_Toc491085974)

[6.4 Location Detail View 34](#_Toc491085975)

[6.5 Presenting Locator Search Results in a List View 35](#_Toc491085976)

[6.6 Filtering Locator Searches 36](#_Toc491085977)

[6.7 Getlist Searches 37](#_Toc491085978)

[6.8 Retrieving Custom Properties 39](#_Toc491085979)

[6.9 Accessing Configuration Parameters 40](#_Toc491085980)

[7 Using Google Analytics 41](#_Toc491085981)

[7.1 Google Analytics SDK Setup 41](#_Toc491085982)

[7.2 Code Samples. 41](#_Toc491085983)

[7.2.1 Adding Screen Tracking 41](#_Toc491085984)

[7.2.2 Adding Event Tracking 42](#_Toc491085985)

[8 DineTime API Integration 43](#_Toc491085986)

[8.1.1 Enhancements to BFConfiguration 45](#_Toc491085987)

[8.1.2 Displaying DineTime Average Wait Times (Powered by Ignite) 45](#_Toc491085988)

[8.1.3 Adding DineTime Waitlist Bookings (Powered by Ignite) 46](#_Toc491085989)

[8.1.4 Modifying DineTime Waitlist Bookings (Powered by Ignite) 47](#_Toc491085990)

[8.1.5 Cancelling DineTime Waitlist Bookings (Powered by Ignite) 48](#_Toc491085991)

[8.2 Managing DineTime Waitlist Bookings 49](#_Toc491085992)

[8.2.1 Caching a Booking 49](#_Toc491085993)

[8.2.2 Fetching a Booking from the Cache 50](#_Toc491085994)

[8.2.3 Removing a Booking from the Cache and Checking for Existing Cached Bookings 51](#_Toc491085995)

[8.2.4 Clearing the Cache 52](#_Toc491085996)

[9 API Reference 53](#_Toc491085997)

[9.1 General Classes 53](#_Toc491085998)

[9.1.1 BFContext Class 53](#_Toc491085999)

[9.2 Model Classes 55](#_Toc491086000)

[9.2.1 BFSearchLocation Class 55](#_Toc491086001)

[9.2.2 BFLocation Class 57](#_Toc491086002)

[9.2.3 BFListOfLocations Class 59](#_Toc491086003)

[9.2.4 BFConfiguration Class 60](#_Toc491086004)

[9.2.5 LocatorConfig Class 61](#_Toc491086005)

[9.2.6 FormdataConfig Class 62](#_Toc491086006)

[9.2.7 GeolocConfig Class 64](#_Toc491086007)

[9.2.8 GeolocsConfig Class 64](#_Toc491086008)

[9.3 Data Provider Classes 65](#_Toc491086009)

[9.3.1 BFMobileSDK Class 65](#_Toc491086010)

[9.3.2 BFLocatorDP Class 67](#_Toc491086011)

[9.3.3 BFConfigurationDP Class 68](#_Toc491086012)

# Version History

|  |  |
| --- | --- |
| **Version** | **Update Description** |
| 1.1.5 | Updated property descriptions on the BFListOfLocations class.  Added method description for the getCustomProperties description on the BFLocation class.  Update SDK Config file descriptions. |
| 1.1.6 | Added the ‘Presenting Locator Search Results in a List View’. Removed duplicated section. |
| 1.1.6 | Added Android documentation. |
| 1.1.6 | Added Googe Analytics API section. |
| 1.1.6 | Updated sections on SDK initialization. |
| 1.1.6 | Added DineTime API guide section. |
| 1.1.6 | Updated information section on Initialize the Brandify Mobile SDK regarding disabling use of IDFA and AAID. |
| 1.1.9 | Added information section on using calling the Getlist API |
| 1.2.0 | Added ability to pre-geocode a partial address |
| 1.2.3 (ios)  1.2.2a (Android) | Adds ability to allow custom key-value pairs in search location object to retrieve custom properties  Removes unused <service> tag reference from AndroidManifest <manifest> structure |

# Introduction

Welcome to the Brandify Mobile SDK. This document contains the Brandify Mobile SDK reference guide as well as a description of a full-featured Brandify Mobile sample application. The SDK is delivered in binary form through either CocoaPods for iOS or through jitpack.io (via Gradle) for Android, while the sample application contains the full source code in Java for Google Android and Swift for Apple iOS. The sample can be utilized as starter kit to develop a new application, or as a sample from which you might copy and modify all components you wish to integrate into your own application.

The SDK and sample applications are fully maintained and with functional expansion and updates on a quarterly basis. It is our goal to maintain the SDK with full backward compatibility.

# SDK Core Functionality

The SDK core functionality for version 1.2.0 consists of

* Locator Search
* Server side configurability in-line with Brandify Web Locator configurability
* User and location tracking to support analytics and the online search to off-line location visit attribution

## Configuration

The Brandify Mobile SDK is configurable. The configuration is driven by a single JSON formatted file used by both the mobile SDK and the Brandify Web Templates.

It is required that all applications using the Brandify Mobile SDK place a file titled sdkconfig.json somewhere in their main application bundle. This means that the file should be included in the build target of your application.

Here is a sample of one sdkconfig.json file.

{

{

"config": {

"demoMode" : 1,

"brandName" : "Joe’s Coffee",

"appkey" : "",

"account" : "joescoffee",

"beaconActionAppkey": "",

"igniteLiteURL" : "",

"sbURL": "https://smilebeacon.com/BrandifyCoupons/api/CouponRest",

"locatorSearchRestURL" : "https://hosted.where2getit.com/rest/locatorsearch",

"getListRestURL" : "https://api.brandify.com/rest/getlist",

"netconfig" : "https://joescoffee.com/sdkconfigs/sdkconfig.json",

"autoStart": true,

"icons" : {

"default" : "https://goo.gl/qgvPvp",

"red" : "https://goo.gl/orj7LE",

"gray" : "https://goo.gl/NqYFwB"

},

"logos":{

"default" : "",

},

"baseStoreImagePathURL" : "",

"defaultStoreImagePathURL" : "",

"sendToEmailImagePathURL" : "https://goo.gl/FrzKTk",

"sendToPhoneImagePathURL" : "https://goo.gl/x564Ad",

"panMilesBeforeRefresh" : 2.0,

"locator": {

"usesGeofences": true,

"formdata": {

"geoip": true,

"dataview": "store\_default",

"limit": 10,

"geolocs": {

"geoloc": [{

"addressline": "",

"country": "",

"latitude": "",

"longitude": ""

}]

},

"radiusuom" : "mile",

"searchradius": "25|50|100|250|500|1000",

"order": "NAME, \_DISTANCE DESC",

"where": {

"and": {

"hasIceCream": {

"eq": "Y"

}

}

}

}

},

"beaconConfig" : {

"listensToBeacons" : true,

"respondsToBeacons" : true

},

"geotrackingConfig" : {

"updateDistanceInMeters" : 2400,

"updateIntervalInMinutes" : 1

}

}

}

When the Brandify Mobile SDK is initialized, it first loads the sdkconfig.json from the main application bundle. Then, the netconfig value (if exists) is used to download the configuration file over the network. Any properties from the downloaded network configuration file will overwrite properties that match the skdconfig.json file in the application bundle. Any properties that do not match between the two configuration files will remain as they were.

The appKey and the locatotorSearchURL properties are used as default values in the BFContext class. These values are set when the SDK is initialized during app launch – within the BFMobileSDK.initialize() call.

The ‘icons’ and the ‘logos’ sections of the configuration file are examples of how the configuration file can be used to customize the graphical mapping results. Locator searches will return an **icon** property on each location result. This property will have the value of ‘default’ or some value configured on the Brandify server for the specified location. Using this location **icon** property value and the **icons** section of the configuration file, a customized mapping pin image can be retrieved and rendered for that specific location. For instance, in the Brandify SDK Sample Application, the configuration file has these values for the icons section.

"icons" : {

"default" : "http://storelocator.mybrand.com/responsive/images/green\_pin.png",

"red" : "http://hosted.where2getit.com/w2gi/images/icons/red-pin-shadow.png",

"gray" : "http://hosted.where2getit.com/w2gi/images/icons/gray-pin-shadow.png"

}

When a location returns the value ‘default’ in its **icon** property, the ‘default’ key in the **icons** section of the configuration can be used to obtain the associated image URL, **http://storelocator.mybrand.com/responsive/images/green\_pin.png**. If the location **icon** property returns the value ‘red’ for a location, then it would obtain the [**http://hosted.where2getit.com/w2gi/images/icons/red-pin-shadow.png**](http://hosted.where2getit.com/w2gi/images/icons/red-pin-shadow.png) image URL.

The defaultStorImagePathURL is an example of an image path that can be used for locations that do not currently have an available image.

The locator section of the configuration file is used to establish default locator search rules.

The configuration file can be viewed as a store of default values. Any property in the configuration can be overridden at runtime with customized values. A good example is the searchradius configuration property. At runtime, under certain circumstances, it is desirable to execute the search using different values than the default. For instance, when panning the map, it is good to provide a searchradius that fits strictly within the view of the current visible map region.

The formdata section of the configuration file contains properties that are explicitly exposed through the SDK. For instance, the *limit* property is exposed through the SDK. However, this section may contain properties that are unknown to the SDK and will be passed onto the Brandify server for processing.

## Customization

Through configuration and the capabilities of the Brandify Mobile SDK, any application can be completely customizable to fit the design goals and brand of your company and provide maximum flexibly to your development team.

Event handler for the Map Kit SDK or geo-location access permission are in the sample code rather than in the Brandify Mobile SDK in order to provide full control to your developers. The SDK includes a 3rd party open source library to cache and lazy-load images, but your developers can easily replace that with your own code or another library. The SDK uses only native iOS and Android code to access the Brandify server, so that it will not interfere with any other libraries you might use to access your own or other 3rd party server code.

# Getting Started in iOS and Android

The sample application is a great place to start. You can use the sample application as a base for building your own branded application or you can use it as a reference. If you are going to use the SDK in your own application, here are the steps you will need to take to use the Brandify Mobile SDK in the iOS and Android development environments.

## Include the Brandify Mobile SDK

### iOS

The iOS Brandify Mobile SDK uses a private CocoaPods to distribute the iOS framework. To include the Brandify Mobile SDK in your iOS project follow these steps.

* Create a file named *Podfile* (if it does not already exist) in the same directory as your XCode project file.
* The following is an example Podfile with the minimal entries necessary to retrieve the 1.1.9 version of the Brandify Mobile SDK.

|  |
| --- |
| platform :ios, '8.0'  use\_frameworks!  source 'ssh://git@github.com/brandify/mobile-sdk-ios-dist.git'  source 'https://github.com/CocoaPods/Specs.git'  target 'BrandifySDKDemo' do  pod 'BrandifyMobileSDK', '1.2.0'  pod 'WebImageFramework', '1.1.0'  end |

* Replace the name of the target (where it says target 'BrandifySDKDemo' do above) with the name of your XCode project.
* It is important to note that when you do include the ‘source’ for the private Brandify Mobile SDK CocoaPods repository, you must also explicitly include the ‘source’ for the standard CocoaPods distribution repository (source 'https://github.com/CocoaPods/Specs.git' ) – if you plan on including other CocoaPods from that standard repository.
* Create an SSH key and provide key information to your Brandify representative. The key will then be added to the GitHub CoacoaPods distribution repository.
  + For specific instructions how to generate an SSH key and add it to your system’s ssh-agent, see <https://help.github.com/articles/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent/>
  + Run the ‘pod update’ command to update your project with the Brandify Mobile SDK framework binary.
  + Note that it may be necessary to run the ‘pod update’ command first to ensure that the Brandify Mobile SDK private repository is up to date on the local system. This will update the private repository and the CocoaPods master repository, so it can take some time.
    - Ensure that when the ‘pod install’ command is run, that the expected Brandify Mobile SDK revision number is noted in the CocoaPods output logging.

After the frameworks are included in your XCode project, the Brandify Mobile SDK classes and methods can be accessed in any code file by importing the BrandifyMobileSDK headers.

|  |
| --- |
| **Swift**  import BrandifyMobileSDK  **Objective-C**  #import <BrandifyMobileSDK/BrandifyMobileSDK-Swift.h> |

### Android

The Brandify Mobile SDK is distributed through jitapck.io. In order to include the Brandify Mobile SDK into your project, add the following lines to the build.gradle file for your project.

|  |
| --- |
| repositories  {  maven  {  url "https://jitpack.io"  credentials {username authToken}  }  } |

The username in the maven credentials is the property title and the authToken referenced in the maven section should be obtained from a Brandify representative and placed in the gradle.properties file. The following is an example.

|  |
| --- |
| authToken=51d7f321bce341d13915ad045e488220191beaab |

Access the Brandify Mobile SDK library components under the com.brandify.BrandifyMobileSDK namespace.

## Initialize the Brandify Mobile SDK

Initializing the SDK will refresh and load the Brandify Mobile SDK configuration cache file. Failing to initialize the Brandify Mobile SDK will yield unpredictable results.

### iOS

In the didFinishLaunchingWithOptions application delegate method (in the Application Delegate class), initialize the Brandify Mobile SDK.

|  |
| --- |
| func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool  {  **BFMobileSDK.initialize(false)**  } |

Note: Unless you specify otherwise when calling this initialize method, default values for the parameter allowsAdvertisingIdTracking will be used and the parameter is not required when calling in Swift (as demonstrated above).  
  
To disable the use of advertising identifiers (i.e. IDFA), include allowsAdvertisingIdTracking in the list of parameters and set it to false. The default is true.

### Android

In the OnCreate method of the Activity defined with the MAIN intent in the AndroidManifest.xml, add the following code.

|  |
| --- |
| BFMobileSDK.initialize(this, false, false); |

Note: For backwards compatibility, the example above will call through to the below method and set allowAdvertisingIdTracking to true as a default. To disable the use of advertising identifiers (i.e. AAID), the following function is provided. Use this alternate method and set allowAdvertisingIdTracking to false.

|  |
| --- |
| public static void initialize(final Context context, boolean usesLocalConfigs, boolean trackLocationInBackground, boolean allowAdvertisingIDTracking) |

It is recommended, but not required to implement the MAIN Activity class from the BFBaseActivity class. This base activity class provides implicit permission checking and, if necessary, permission requesting during the initialization process. If the BFBaseActivity class is not used to implement the MAIN Activity class, then it is required to check location permissions and, if necessary, *request* location permissions prior to initializing the Brandify Mobile SDK.

It is also important to note that when the location permissions for the demo application (or any other Android application) change while the application is open, the application will be restarted by the operating system. When this happens, the Brandify Mobile SDK must also be initialized. The best way to ensure that this always happens, the initialization call should occur in the MAIN Activity class as previously noted and the instance type should be set to ‘singleTop’ in the AndroidManifest.XML file.

|  |
| --- |
| <activity  android:launchMode="singleTop"  android:name=".Home.HomeActivity"  android:label="@string/app\_name">  <intent-filter>  <action android:name="android.intent.action.MAIN"/>  <category android:name="android.intent.category.LAUNCHER"/>  </intent-filter>   </activity> |

## Building the Distributed iOS Application Binary

The Brandify Mobile SDK framework included in the sample application is considered a universal or fat framework. This means that it includes both x86 and ARM architectures.

*NOTE: If it is ever necessary to determine which architectures are included in any iOS binary, then use a command line tool on the macOS and navigate into the app for framework and execute the following command on the particular binary.*

*lipo –info <binary-name>*

*For instance, using the BrandifyMobileSDK.framework file located in the demo application, the following results from the ‘lipo –info BrandifyMobileSDK’ command:*

Architectures in the fat file: BrandifyMobileSDK are: i386 x86\_64 armv7 arm64

The advantage of a fat framework is that it will run on both iOS simulators and iOS devices. However, the iTunes App Store requires that applications only include the ARM architectures when submitting to the App Store. Use the following ‘Run Script’ to strip the x86 architectures prior to submission.

|  |
| --- |
| APP\_PATH="${TARGET\_BUILD\_DIR}/${WRAPPER\_NAME}"  # This script loops through the frameworks embedded in the application and  # removes unused architectures.  find "$APP\_PATH" -name 'BrandifyMobileSDK.framework' -type d | while read -r FRAMEWORK  do  FRAMEWORK\_EXECUTABLE\_NAME=$(defaults read "$FRAMEWORK/Info.plist" CFBundleExecutable)  FRAMEWORK\_EXECUTABLE\_PATH="$FRAMEWORK/$FRAMEWORK\_EXECUTABLE\_NAME"  echo "Executable is $FRAMEWORK\_EXECUTABLE\_PATH"  EXTRACTED\_ARCHS=()  for ARCH in $ARCHS  do  echo "Extracting $ARCH from $FRAMEWORK\_EXECUTABLE\_NAME"  lipo -extract "$ARCH" "$FRAMEWORK\_EXECUTABLE\_PATH" -o "$FRAMEWORK\_EXECUTABLE\_PATH-$ARCH"  EXTRACTED\_ARCHS+=("$FRAMEWORK\_EXECUTABLE\_PATH-$ARCH")  done  echo "Merging extracted architectures: ${ARCHS}"  lipo -o "$FRAMEWORK\_EXECUTABLE\_PATH-merged" -create "${EXTRACTED\_ARCHS[@]}"  rm "${EXTRACTED\_ARCHS[@]}"  echo "Replacing original executable with thinned version"  rm "$FRAMEWORK\_EXECUTABLE\_PATH"  mv "$FRAMEWORK\_EXECUTABLE\_PATH-merged" "$FRAMEWORK\_EXECUTABLE\_PATH"  done |

## Configure the Brandify Mobile SDK Inside your Mobile Application

As discussed in the section 3.1, the Brandify Mobile is configurable. The first step of configuration is to make sure your appKey and locatoreSearchRestURL properties are set correctly. These two properties ensure that locator searches execute correctly through the SDK.

Configure the UI elements such as map pin images, logos images, and images specific to your brand. Establish store locator search defaults, look and feel controls, and map handling controls such as how many miles to pan before another locator search is executed.

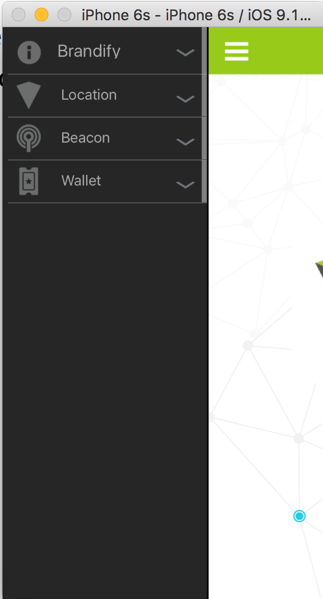
## The Brandify Sample Application

The sample application demonstrates how the Brandify Mobile SDK framework can be included in an iOS or Android application and integrated with popular mapping frameworks, such as the iOS MapKit framework and the Google Maps API, to quickly develop and deliver robust location aware applications.

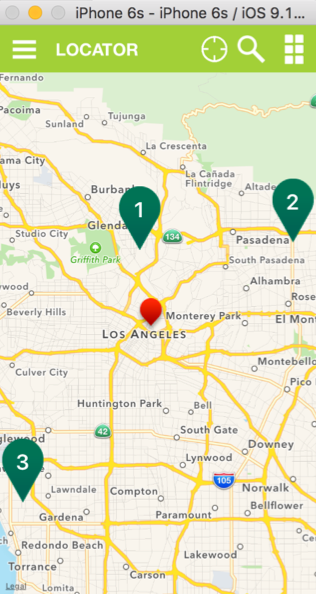
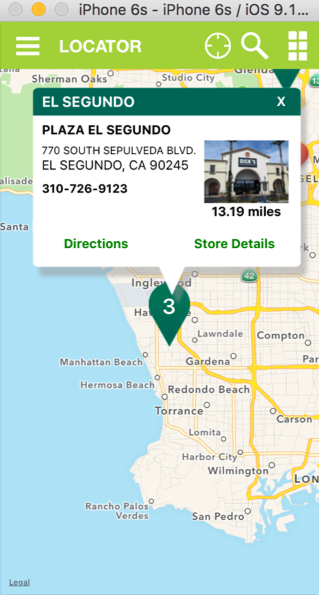
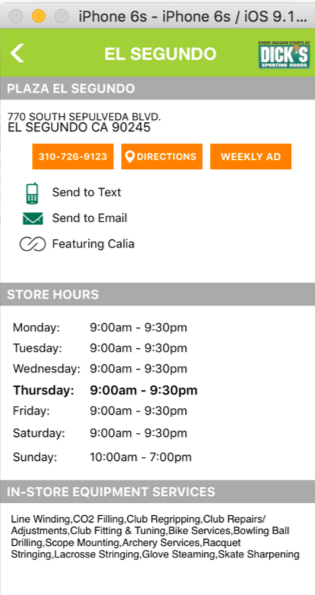
The Brandify sample application contains the following features.

* Navigation
* Location Functionality
* Beacon Functionality
* Wallet Functionality

For *navigation*, the sample application employs a left menu that either slides in from the left in response to a right swipe or by selecting the ‘hamburger’ menu icon. The menu can be closed by right swiping, by selecting a menu item, or by selecting outside of the menu area.

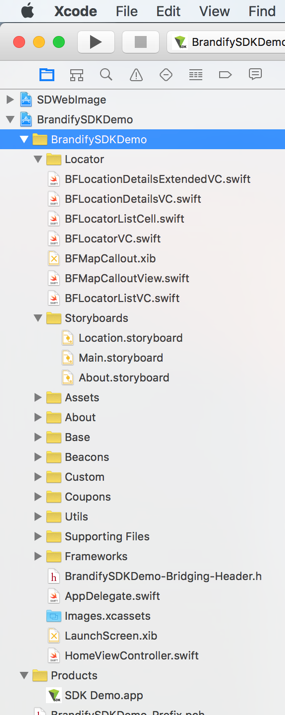


Location functionality consists of locator searches, mapped or list results view, and the ability to view details for any returned location

The sample application has both Beacon and Wallet functionality. Both feature sets are rudimentary and will be updated in future releases of the sample application.

# Analyzing the iOS Sample Application



The core of the Brandify Mobile SDK and iOS MapKit integration can be found in the BFLocatorVC.swift file. BFLocatorVC.swift is an iOS View Controller and is connected to the LocatorVC scene in the Location.storyboard file.

The custom callout view used in the Brandify Mobile SDK and iOS MapKit integration is defined by the BFMapCalloutView.swift file and the BFMapCallout.xib file.

Users can select 'Store Details' in the custom callout view to see more location details. The BFLocationDetailsVC.swift, BFLocationDetailsExtendedVC.swift, and the LocationDetailsVC scene in the Location.storyboard are used to produce the 'Store Details' view. The LocationsDetailsVC scene contains an embedded LocationDetailsExtendedVC scene. This provides the LocationDetailsVC scene with the freedom to be designed and connected without the restrictions imposed by a table view controller.

**iOS Storyboards:**

Location.storyboard is the primary storyboard for the Brandify Mobile SDK integration. In general, the iOS storyboards decouple the view design from the data. The storyboard designer includes many UI tools and access to UI properties that allow for the interface to be professionally designed without having to write code. All UI elements can be dynamically updated using IBOutlet connections, which connect UI elements to the code and ultimately to the data. Actions can be captured from the storyboard UI elements and sent to the code using IBAction connections. For instance, a button may send an action request to a method in the code by using an IBAction connection.

## Locator Searches

The SDK provides the ability to find store locations based on partial or full address information or by the current geo-position of the device. The following describes how to use the SDK to execute a locator search using the iOS Swift language.

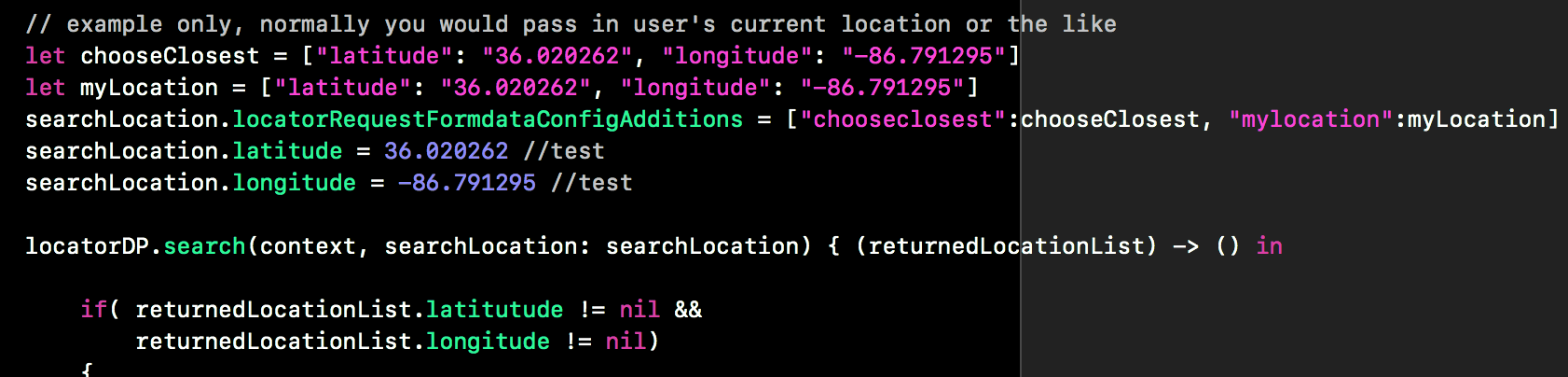
* Create a BFContext instance. The BFContext instance holds the connection information to execute the search call.

**var context = BFContext()**

* Create a BFSearchLocation instance. The BFSearchLocation instance holds the rules for the search. In this snippet, we only instantiate the instance and we do not set any properties that would bound the search. In this case, the locator search will use the IP address of the device to determine the current location and center the search based on that information.

**var searchLocation = BFSearchLocation()**

* Create a BFLocatorDP instance. The BFLocatorDP instance is a data provider that handles the search execution over the network and the transfer of data from the server to the client.
* Note that the search() function on BFLocatorDP utilizes the formData property from sdk\_config.json to form the API request. For clients requiring to dynamically modify this before sending a location search request, pass a dictionary containing the desired key-value pairs for the search request by setting locatorRequestFormdataConfigAdditions on BFSearchLocation as shown in the example below. If it is expected that passing in custom request parameters in this way should cause the API to return custom properties, see the section entitled Retrieving Custom Properties for details on how to read these values.



**var locatorDP = BFLocatorDP()**

* Execute the locator search by call the search() method on the BrandifyLocatorDP instance.
* **New to v 1.2.0**, you can now pass in true for the shouldGeocode parameter to ensure that actual locations are returned to the locator for partial address searches. Previously, if the search term did not contain geocode information alongside the search term query, it could return suggested locations for the search term instead. For backwards compatibility in Swift, the default of the shouldGeocode property is set to false. This allows the search() to continue working OOTB with your existing locator implementations. If running Objective-C code, this method call will need to be updated to the new method signature.

**locatorDP.search(context, searchLocation, shouldGeocode (listOfLocations) -> () in**

**{**

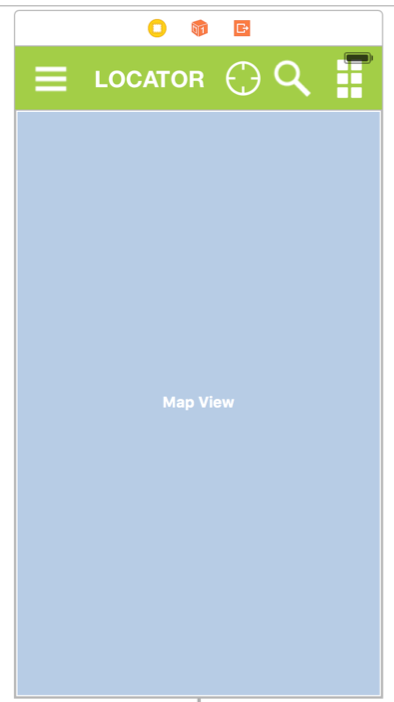
**//process the returned list of locations**

**}**

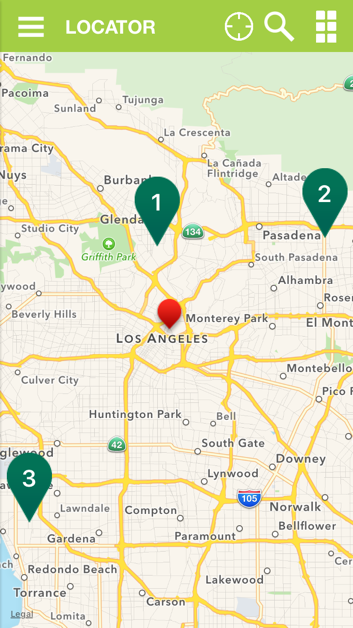
* After the search executes, a list of locations will be returned in a completion block. The list of locations is a BFListOfLocations instance. On the BFListOfLocations instance itself is a coordinate that provides the latitude and longitude of the center of the search along with a collection of BFLocation instances – if any. Each location in the list provides a number of properties describing the location such as its coordinates, name, phone number, and address. This information is suitable for integration with integration with any graphical geo mapping development kit such as the iOS Map Kit.

## Integrate with the iOS Map Kit

Create a storyboard scene and add an MKMapView control.



Create a UIViewController class that connects an IBOutlet to the MKMapView control. Make the UIViewController class the delegate for MKMapView callback methods. Within that UIViewController class, use the results of locator searches to display map pins according to each returned location.

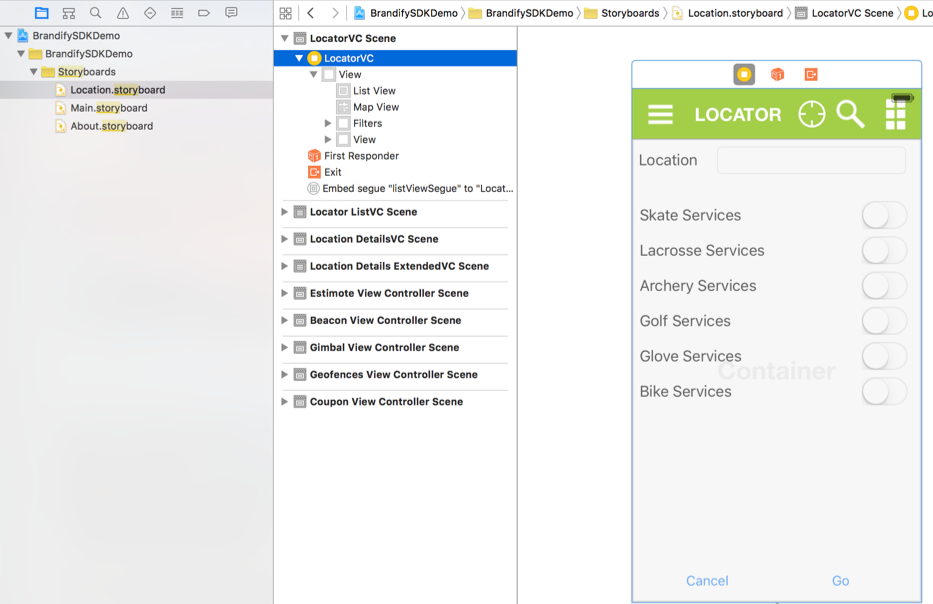
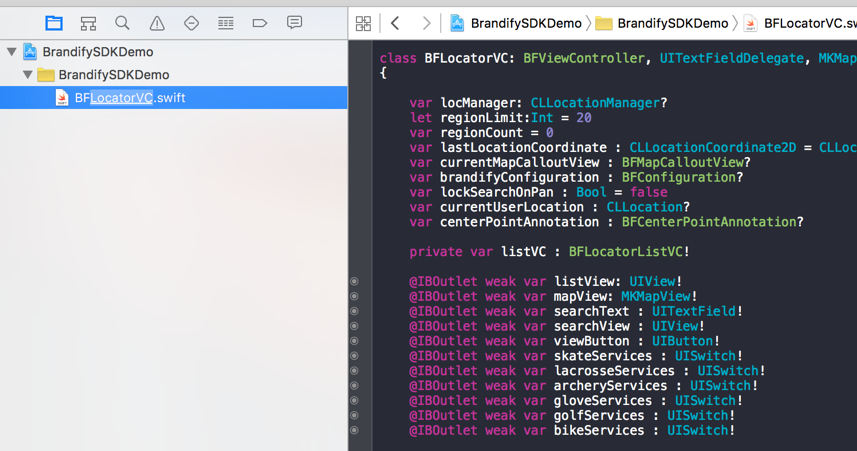


Customize map pins and callouts by creating your own MKMap Kit derived classes. See the sample application BFLocatorVC.swift file for sample derived classes and specific examples.

## Locator Searches using The Brandify Mobile SDK and iOS MapKit Integration

This section looks at the details of the Brandify Mobile SDK and the iOS MapKit integration located in the BFLocatorVC.swift file and the LocatorVC scene in the Location.storyboard file.

The LocatorVC scene contains a view based on the MKMapView. The MKMapView is connected to the BFLocatorVC class through the 'mapView' outlet.



Much of the integration relies on responding to MKMapView delegate methods. The delegates are routed to the BFLocatorVC class by setting the mapView.delegate equal to the instance of the BFLocatorVC. When inspecting the BFLocatorVC.swift file, all of the MKMapView delegates can be identified by the 'mapView' methods. For example, the 'viewForAnnotation' MKMapView delegate can be identified by this method signature.

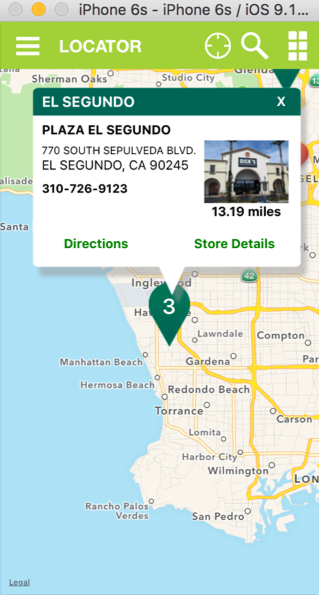
**func mapView(mapView: MKMapView, viewForAnnotation annotation: MKAnnotation) -> MKAnnotationView?**

All Brandify Mobile SDK locator searches are executed in the getTargetLocations() method in the BFLocatorVC class. However, there are four areas that control how and when the locator searches will execute.

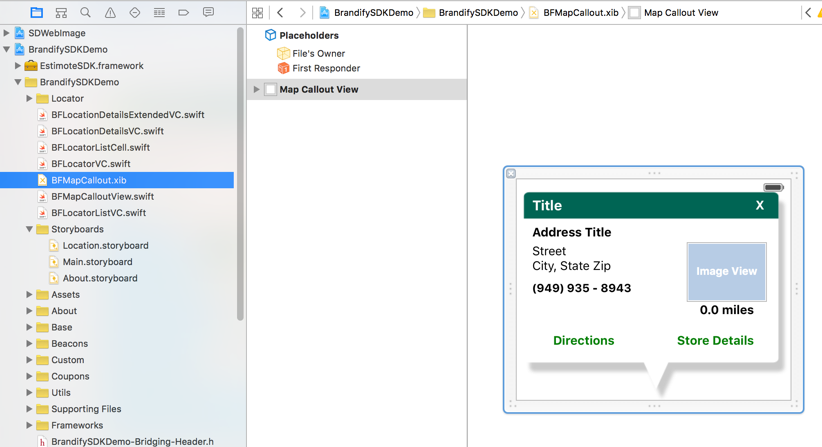
* **viewDidLoad()** - This is called when the view loads. By default, the search will attempt to search by the current location of the mobile device. If the current location is not available, then the IP address of the device will be used resolve the current location. The current location of the device may be unavailable if the user chose not to share the location of the device or because the current location could not be established soon enough. Note that IP address resolution may not be accurate in cases where the IP address is obtained through a non-wifi connection (4G for instance).
* **mapView regionDidChangeAnimated delegate method** - This is used control locator searches when the map is panning.
* **restToUserLocation**() - When the user selects to return to their current location in the map.
* **runSearch**() - An IBAction method used to execute searches through a view button.

## Custom Callout

The callout in the sample application has been customized.



All of the UI customization is located in the BFMapCallout.xib file.



The customized callout is implemented using a simple UIView and is initiated during the didSelectAnnotationView delegate method. See the showMapCallout() method in the BFLocatorVC.swift for programming details.

## Location Detail View

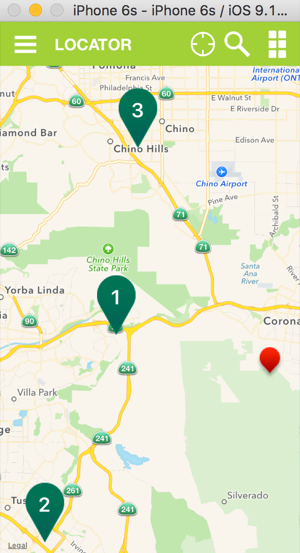
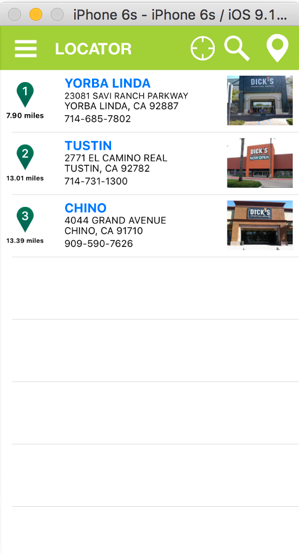
The “Store Details” button can be selected within the customized callout to open an expanded view of the store. The UI portion of the store details view is driven almost entirely by the storyboard. Therefore, it is trivial to rearrange the different detail areas to match a different design.



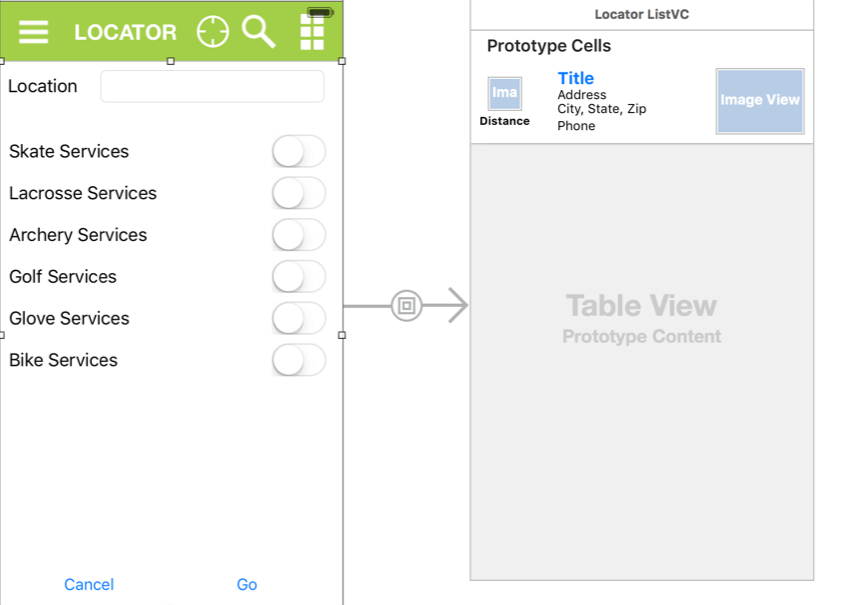
The UI elements are connected through IBOutlets to the BFLocatorVC class in the BFLocatorVC.swift file.

## Presenting Locator Search Results in a List View

In addition to displaying locator search results in a map view, it is also possible to render the results in a ‘list view’ style. In the sample application, the map view is the default view. The ‘list view’ can be presented by selecting the ‘list view’ icon in the upper right corner. Return to the ‘map view’ by tapping the map pin icon in the upper right corner.



The BFLocatorListVC.swift and the BFLocatorListCell.swift implement the necessary classes to render the list view. The UI portion of the list view is contained within the LocatorVC storyboard scene and uses a ‘Container View’.

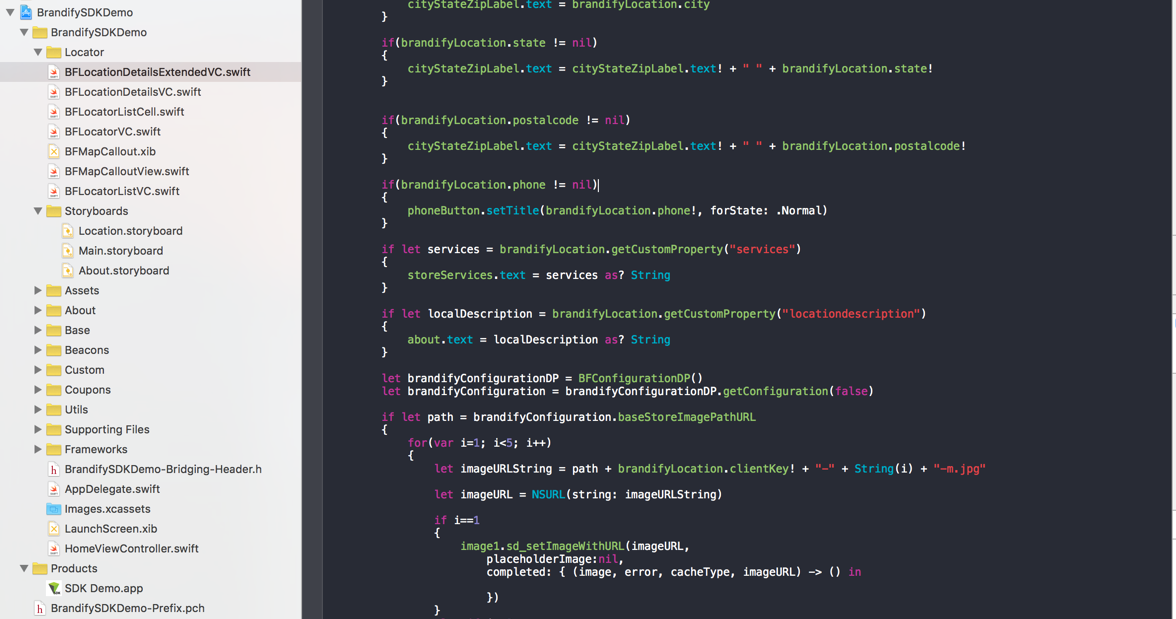


## Filtering Locator Searches

Included in the sample application is the ability filter locator searches based on specific user input. A method titled getFilters() in the BFLocatorBV class inspects six UISwitch controls on the Filters view in the LocatorVC scene (Location.storyboard) returns a dictionary of properties based on the state of those controls. The resulting dictionary of properties is used to update the BFSearchLocation ***filterProperties*** member prior to executing searches. When searches are executed through the SDK, the filterProperties dictionary will be inspected and If matches are found between the filterProperties and the ‘where’ section in the configuration file, then those properties in the configuration file will be updated with the values in the ‘filterProperties’ dictionary.

## Retrieving Custom Properties

There are a number of general properties on the BFLocation class, locationName for instance. Depending on the implementation, an agency may have several custom properties for their locations. These properties will not be found on the BFLocation class. A utility method titled getCustomProperty() on the BFLocation class is available to obtain the value of those custom properties. An example of this can be found in the viewDidLoad() method in the BFLocationDetailsExtendedVC class.

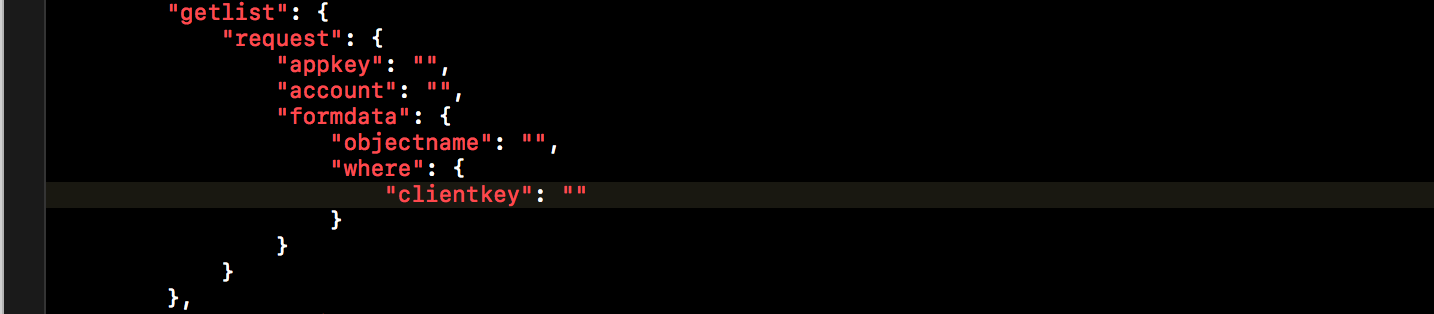


## Getlist Searches

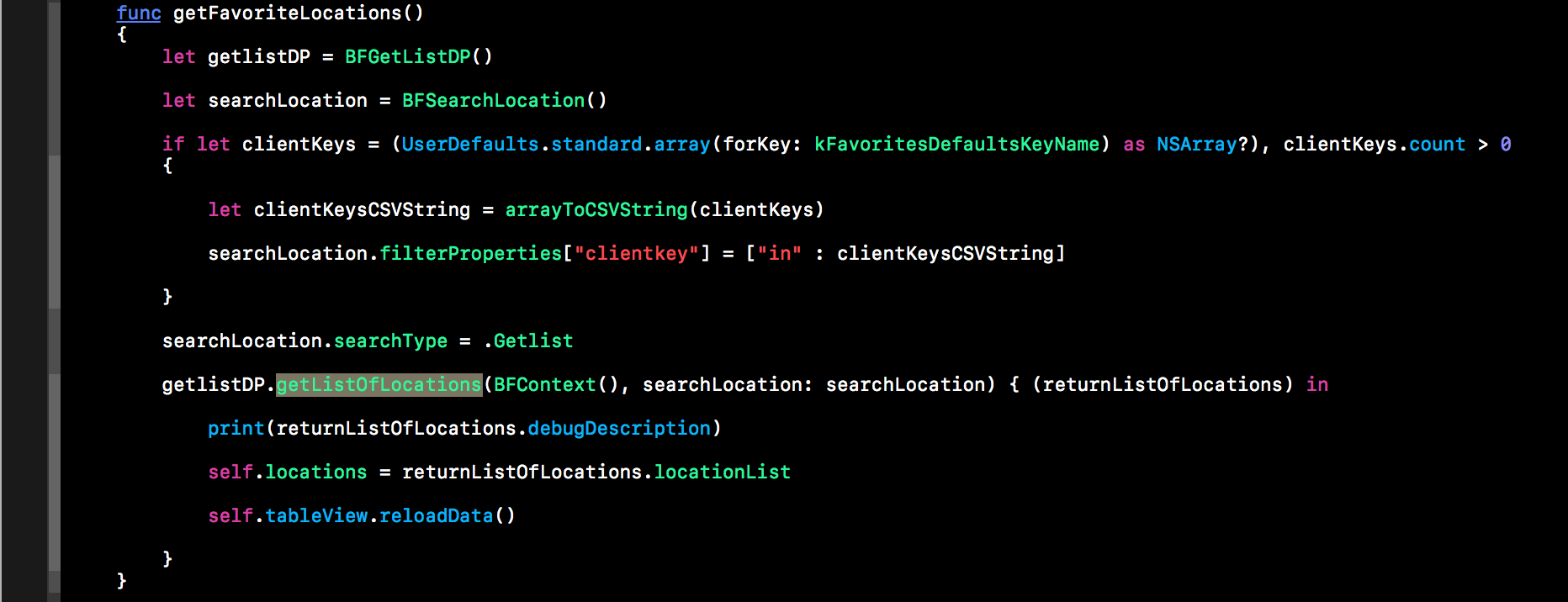
The sample application contains a mock “My Favorite Locations” feature. Note that for the purposes of the example, we are saving the location identifiers to NSUserDefaults. This can persist data between app launches and even updates, but this data would be lost if the user were to delete the application. In a production application, these values should be saved in a more robust fashion.

The BFFavoriteLocationsTableViewController class contains a method called getFavorites(). This method retrieves one or more location identifiers from NSUserDefaults and then calls the Brandify API to retrieve the most recent location information. This is accomplished via the getListOfLocations() method of the BFGetlistDP class.

The getListOfLocations() method takes a BFContext and a BFSearchLocation. The “searchType” property of BFSearchLocation is set to BFSearchType.Getlist. For backwards compatibility, the default of the searchType property is BFSearchType.Locator. This allows BFSearchLocation to continue working OOTB with your existing locator implementations. Setting the “searchType” to .Getlist will cause the request query to be based off of the “getlist” dictionary in the sdkconfig.json file.

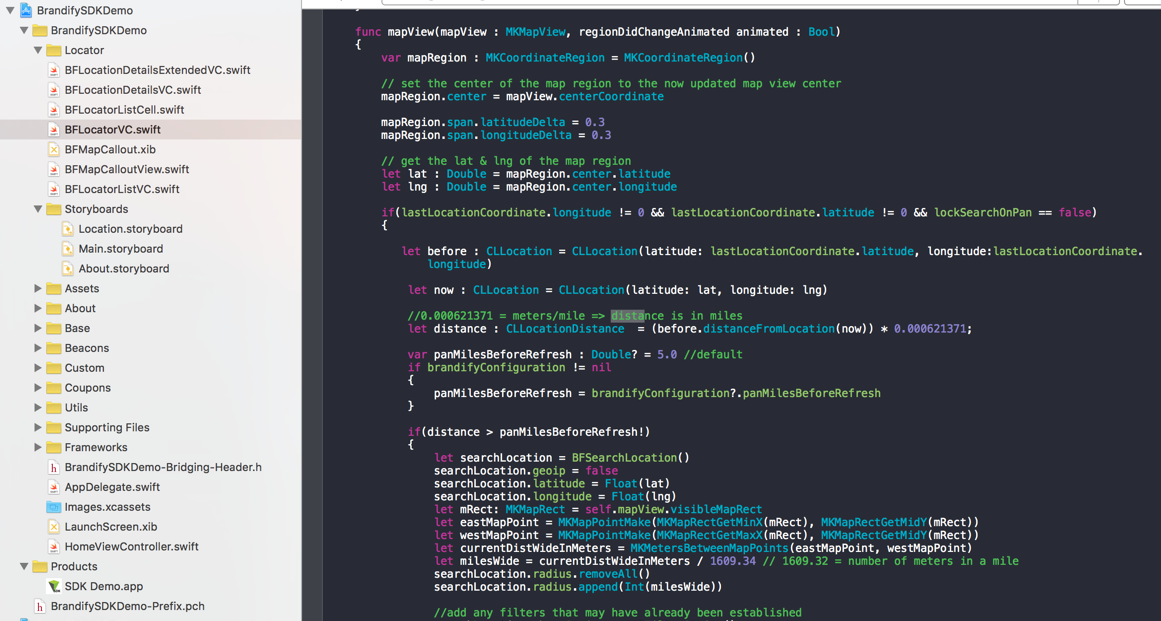


The property or properties to be filtered over in your getlist search must be preplanned, but the values for the filters can be provided dynamically just as described in section 5.7 Filtering Locator Searches. In the sample application, “clientkey” is used. The value for “clientkey” is provided via the filterProperties property on BFSearchLocation. The “in” operator is used to allow flexibility to request one or more locations at a time. Note that when passing an array of values, a comma-separated string representation must be used as shown below.



## Accessing Configuration Parameters

The getConfiguration() method on BFConfigurationDP class is used to retrieve Brandify Mobile SDK configuration information. This method takes a single Boolean parameter, forceRefresh, and returns an instance of the BFConfiguration class. If forceRefresh is true, the configuration will be retrieved from local cache, otherwise the BFConfigurationDP class will attempt to retrieve the configuration first from the application bundle, then from the network. An example usage of the configuration class can be found in the BFLocatorVC class in the regionDidChangeAnimated delegate method. In this method, the configuration is used to control the distance a map must pan before executing a new locator search.

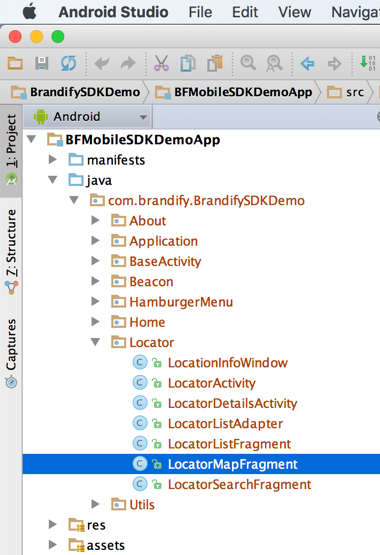
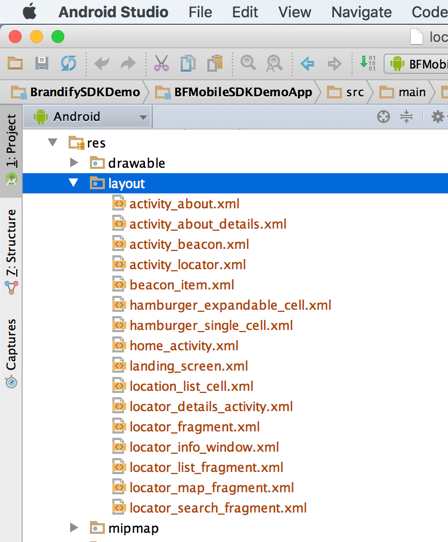


Note that in the sample application, in the AppDelegate.swift file, the configuration is always refreshed on launch of the application in response to the BFMobileSDK.initialize() call.

# Analyzing the Android Sample Application

The sample application demonstrates how the Brandify Mobile SDK framework can be included in an Android application and integrated with other frameworks, such as the Google Maps API, to quickly develop and deliver robust location aware applications.

A powerful feature of the SDK is its ability to integrate with the Google Maps API. The com.brandify.BrandifySDKDemo.Locator and the res.layouy groups in the Android Studio sample application project contain the most significant files for the Google Maps integration.



The LocatorActivity class serves as the controller for the locator. It controls the map view (LocatorMapFragment), list view (LocatorListFragment/LocatorListAdapter), the search view (LocatorSearchFragment), and the callout view (LocationInfoWindow).

When the callout view is touched, the LocatorDetailsActivity class is instantiated and presented.

The following table describes association of activities/fragments to the XML view resource file.

|  |  |
| --- | --- |
| **Activity/Fragment** | **XML View Resource** |
| LocatorMapFragment | locator\_map\_fragment.xml |
| LocatorListFragment | locator\_list\_fragment.xml |
| LocatorSearchFragment | locator\_search\_fragment.xml |
| LocationInfoWindow | locator\_info\_window.xml |
| LocatorDetailsActivity | locator\_details\_activity.xml |

## Locator Searches

The SDK provides the ability to find store locations based on partial or full address information or by the current geo-position of the device. The following describes how to use the SDK to execute a locator search using the Java programming language in Android.

* Create a BFContext instance. The BFContext instance holds the connection information to execute the search call.

**BFContext context = BFContext();**

* Create a BFSearchLocation instance. The BFSearchLocation instance holds the rules for the search. In this snippet, we only instantiate the instance and we do not set any properties that would bound the search. In this case, the locator search will use the IP address of the device to determine the current location and center the search based on that information.

**BFSearchLocation searchLocation = BFSearchLocation();**

* Create a BFLocatorDP instance. The BFLocatorDP instance is a data provider that handles the search execution over the network and the transfer of data from the server to the client.
* Note that the search() function on BFLocatorDP utilizes the formData property from sdk\_config.json to form the API request. For clients requiring to dynamically modify this before sending a location search request, pass a dictionary containing the desired key-value pairs for the search request by setting locatorRequestFormdataConfigAdditions on BFSearchLocation as shown in the example below. If it is expected that passing in custom request parameters in this way should cause the API to return custom properties, see the section entitled Retrieving Custom Properties for details on how to read these values.



**BFLocatorDP locatorDP = BFLocatorDP();**

* Execute the locator search by call the search() method on the BrandifyLocatorDP instance. A list of locations is returned through the BFRunnable parameter of the search() method.

**locatorDP.search(context, searchLocation, new BFRunnable<BFListOfLocations>()  
{  
 @Override  
 public void run()  
 {**

**BFListOfLocations listOfLocations = this.model;**

**//iterate through locations…**

**}**

**}**

## Integrating with the Google Maps API

The Android sample application integrates with the Google Maps API through a fragment titled LocatorMapFragment. In this fragment, the Brandify SDK is used to execute locator searches. Using an instance of **com.google.android.gms.maps.GoogleMap**, the results are added as **markers** to the GoogleMap instance.

**locator\_map\_fragment.xml (snippet…)**

<**fragment xmlns:android="http://schemas.android.com/apk/res/android"  
 xmlns:tools="http://schemas.android.com/tools"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 android:id="@+id/map"  
 tools:context=".MapsActivity"  
 android:name="com.google.android.gms.maps.SupportMapFragment"  
 android:tag="map"  
 tools:layout="@layout/activity\_locator"**/>

**Snippet from the LocatorMapFragment…**

**private void** setUpMap()  
{  
 *mapFragment*.getMapAsync(**new** OnMapReadyCallback()  
 {  
 @Override  
 **public void** onMapReady(GoogleMap googleMap)  
 {  
 **//set the GoogleMap instance into our class level variable…**

*mMap* = googleMap;  
 *mMap*.setInfoWindowAdapter(**new** GoogleMap.InfoWindowAdapter()  
 {

}

}

}

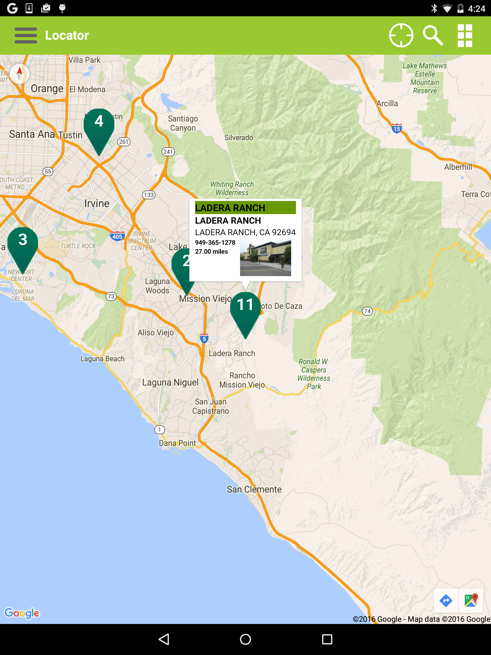
}

**Snippet from the LocatorMapFragment (iterating through locator search results)…**

**for** (BFLocation location : **returnedLocationList**.locationList)  
{  
 LatLng pos = **new** LatLng(location.latitude, location.longitude);  
 Marker marker = *mMap*.addMarker(…);  
}

## Custom Callout

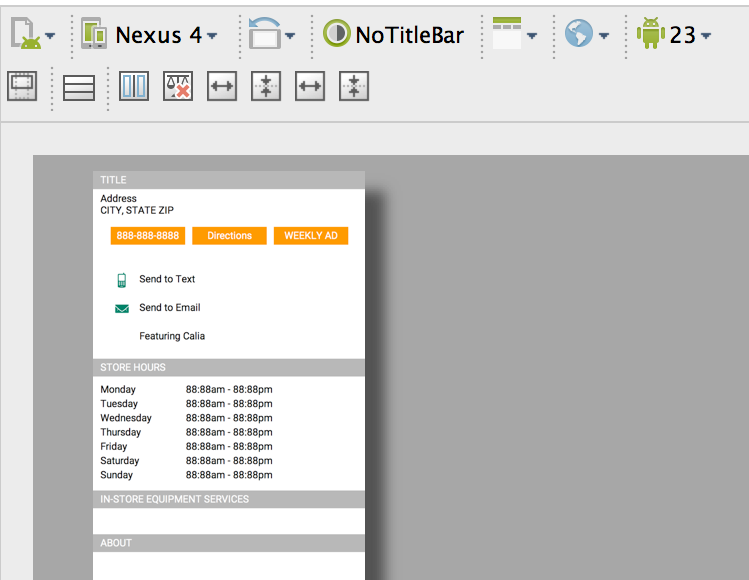
The callout in the sample application has been customized.



All of the UI customization is located in the locator\_info\_window.xml and the LocatorInfoWindow class The customized callout is implemented using an InfoWindowAdapter.

## Location Detail View

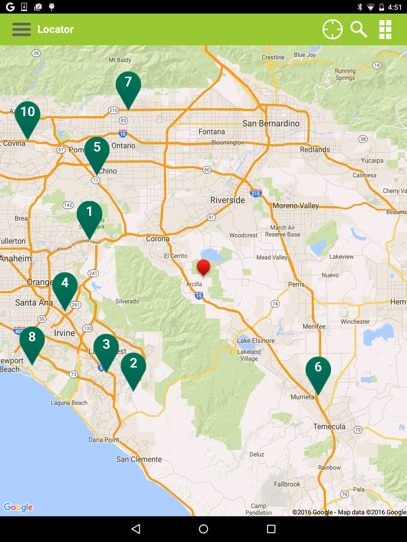
Touching the custom callout will launch the Location Details Activity. The UI portion of the location details view is driven almost entirely by the XML view resource file. Therefore, it is trivial to rearrange the different detail areas to match a different design.



## Presenting Locator Search Results in a List View

In addition to displaying locator search results in a map view, it is also possible to render the results in a ‘list view’ style. In the sample application, the map view is the default view. The ‘list view’ can be presented by selecting the ‘list view’ icon in the upper right corner. Return to the ‘map view’ by tapping the map pin icon in the upper right corner.

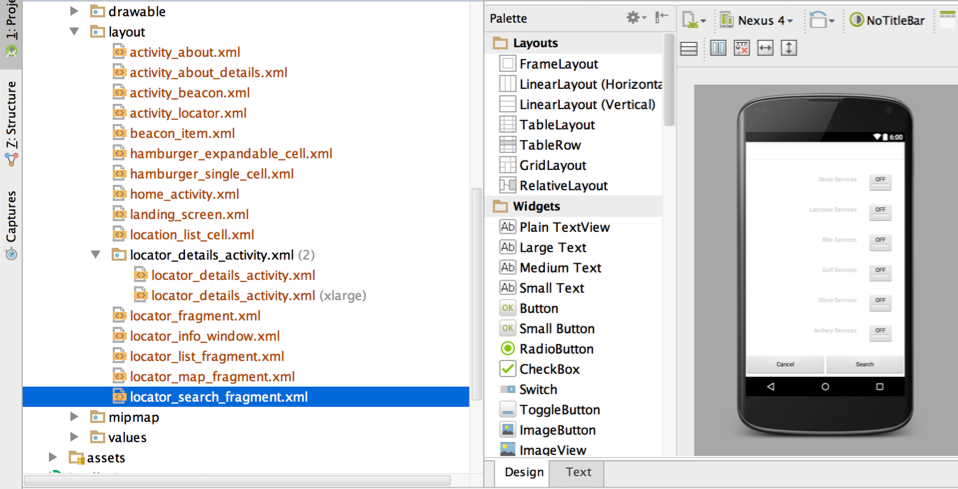




The LocatorListFragment, LocatorListAdapter, and the locator\_list\_fragment.xml contain all for the logic and UI elements for rendering the list view.

## Filtering Locator Searches

Included in the sample application is the ability filter locator searches based on specific user input. A method titled getFilters() in the LocationSearchFragment class inspects six UISwitch controls on the locator\_search\_fragment.xml resource view. A HashMap dictionary is returned. The resulting dictionary of properties is used to update the BFSearchLocation ***filterProperties*** member prior to executing searches. When searches are executed through the SDK, the filterProperties dictionary will be inspected and If matches are found between the filterProperties and the ‘where’ section in the configuration file, then those properties in the configuration file will be updated with the values in the ‘filterProperties’ dictionary.



## Getlist Searches

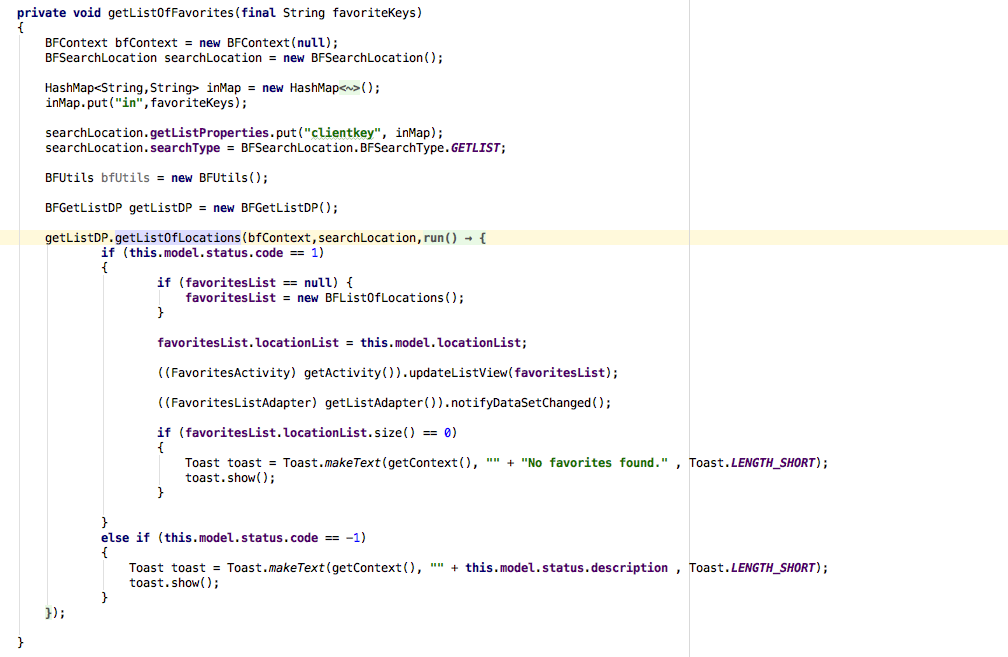
The sample application contains a mock “My Favorite Locations” feature. Note that for the purposes of the example, we are saving the location identifiers directly on the device’s internal storage which by default, are private and can’t be accessed by other external applications. This can persist data between app launches and even updates, but this data would be lost if the user were to delete the application. In a production application, these values should be saved in a more robust fashion.

The FavoritesListFragment class contains a method called fetchFavorites(). This method retrieves one or more location identifiers from the internal storage using a utility method and then calls the Brandify API to retrieve the most recent location information. This is accomplished via the getListOfLocations() method of the BFGetListDP class.

The getListOfLocations() method takes a BFContext and a BFSearchLocation. The “searchType” property of BFSearchLocation is set to BFSearchType.GETLIST. For backwards compatibility, the default of the searchType property is BFSearchType.LOCATOR. This allows BFSearchLocation to continue working OOTB with your existing locator implementations. Setting the “searchType” to .*GETLIST* will cause the request query to be based off of the “getlist” dictionary in the sdkconfig.json file.



The property or properties to be filtered over in your getlist search must be preplanned, but the values for the filters can be provided dynamically just as described in section 6.6 Filtering Locator Searches. In the sample application, “clientkey” is used. The value for “clientkey” is provided via the filterProperties property on BFSearchLocation. The “in” operator is used to allow flexibility to request one or more locations at a time. Note that when passing an array of values, a comma-separated string representation must be used as shown below.



## Retrieving Custom Properties

There are a number of general properties on the BFLocation class, locationName for instance. Depending on the implementation, an agency may have several custom properties for their locations. These properties will not be found on the BFLocation class. A utility method titled getCustomProperty() on the BFLocation class is available to obtain the value of those custom properties. An example of this can be found in the setStoreInformation() method in the LocationDetailsActivity class.

**private void** setStoreInformation()  
{  
 **this**.**storeEquipText** = (TextView)findViewById(R.id.***equiment\_services\_text***);  
 String services = **selectedLocation**.getCustomProperty(**"services"**).toString();

**...**

}

## Accessing Configuration Parameters

The getConfiguration() method on BFConfigurationDP class is used to retrieve Brandify Mobile SDK configuration information. This method takes a single Boolean parameter, forceRefresh, and returns an instance of the BFConfiguration class. If forceRefresh is true, the configuration will be retrieved from local cache, otherwise the BFConfigurationDP class will attempt to retrieve the configuration first from the application bundle, then from the network. An example usage of the configuration class can be found in the LocatorMapFragment class in the setUpMat() method (onCameraChangeListener callback). In this method, the configuration is used to control the distance a map must pan before executing a new locator search.

**double** panMilesBeforeRefresh = 5.0;  
**if** (**bfConfig** != **null**)  
{  
 panMilesBeforeRefresh = **bfConfig**.panMilesBeforeRefresh;  
}  
**if** (distance > panMilesBeforeRefresh)  
{

...

}

Note that in the sample application, in the HomeActivity class, the configuration is always refreshed on launch of the application in response to the BFMobileSDK.initialize() call.

# Using Google Analytics

## Google Analytics SDK Setup

To successfully utilize Google Analytics platform and tools, follow the setup instruction provided at Google's Developer Analytics page, <https://developers.google.com/analytics/> which will provide instructions on properly configuring the targeted platform for using Google Analytics.

## Code Samples.

The following are code examples for adding tracking to specific events using Google Analytics SDK.

### Adding Screen Tracking

#### iOS

Add header to file to desired screen to be tracked.

|  |
| --- |
| **Swift**  import Google  **Objective C**  #import <Google/Analytics.h> |

It is recommended that inside the viewWillAppear method, create a reference to the Google Analytics default tracker and configure the object with the screen name and then executed in the following manner.

|  |
| --- |
| let tracker = GAI.sharedInstance().defaultTracker  tracker.set(kGAIScreenName, value: screenName)    let builder = GAIDictionaryBuilder.createScreenView()  tracker.send(builder.build() as [NSObject : AnyObject]) |

#### Android

Open the Activity that that is going to be tracked by Google analytics. You could also track a Fragment, but ensure that it correctly represents a screen view.

Override the onCreate method of the Activity or Fragment you want to track to obtain the shared Tracker instance:

|  |
| --- |
| // Obtain the shared Tracker instance. AnalyticsApplication application = (AnalyticsApplication) getApplication(); mTracker = application.getDefaultTracker(); |

Override the appropriate method, such as onResume for an Acitivity or onPageSelected for a ViewPager to log when the screen changes.

|  |
| --- |
| Log.i(TAG, "Setting screen name: " + name); mTracker.setScreenName("Image~" + name); mTracker.send(new HitBuilders.ScreenViewBuilder().build()); |

### Adding Event Tracking

To track specific event or user actions within the app, use a Google event to record and describe the interaction. The following shows an example of an event being used to track a button click.

#### IOS

Add header to file to desired screen to be tracked.

|  |
| --- |
| **Swift**  import Google  **Objective C**  #import <Google/Analytics.h> |

Inside the method that captures the specific button click being measured, a tracker object must be created, configured and executed in the following manner.

|  |
| --- |
| let tracker = GAI.sharedInstance().defaultTracker    // Optional: configure the screen name to link event to a screen if event occurs in a screen that is not already being tracked.  tracker.set(kGAIScreenName, value: screenName)    tracker.send(GAIDictionaryBuilder.createEventWithCategory("UIAction", action: "buttonPress", label: nameOfButton, value: value).build() as [NSObject : AnyObject]) |

#### Android

To send an event to Google Analytics, use [HitBuilders.EventBuilder](https://developer.android.com/reference/com/google/android/gms/analytics/HitBuilders.EventBuilder.html) and send the hit, as shown in this example:

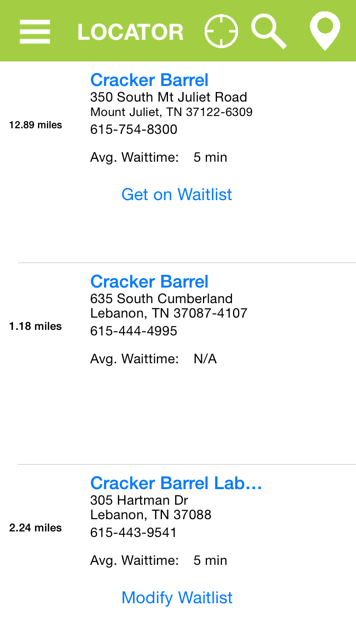
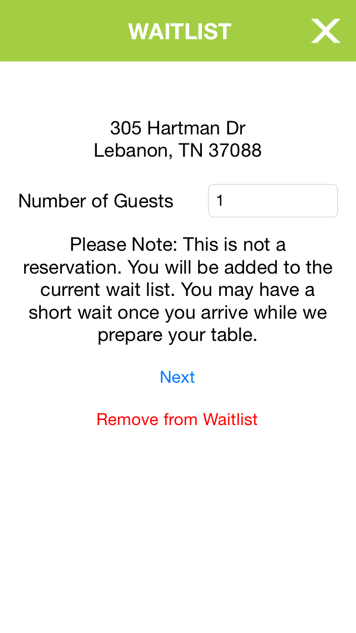
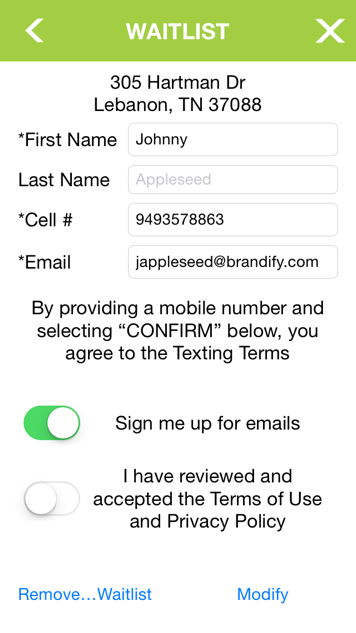
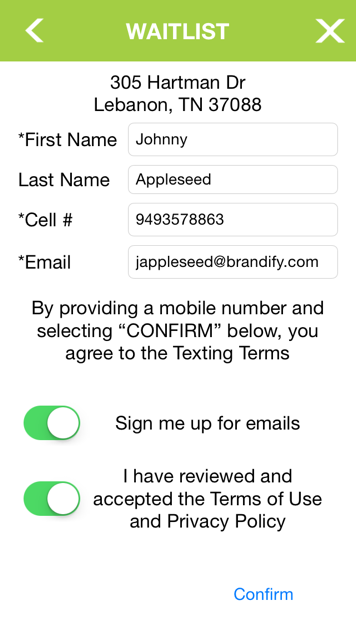
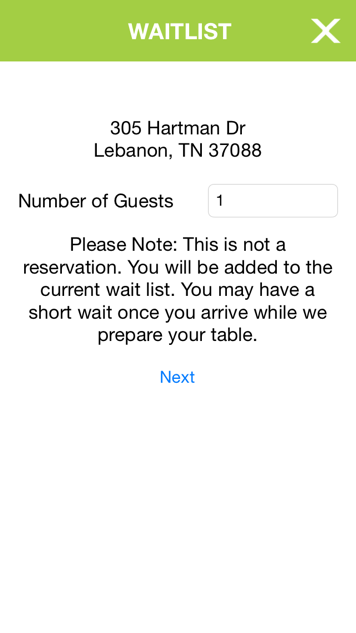
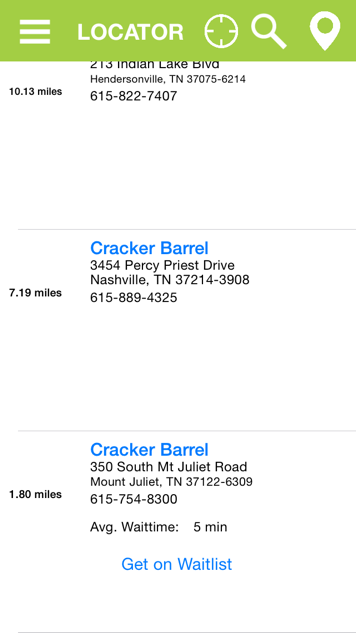
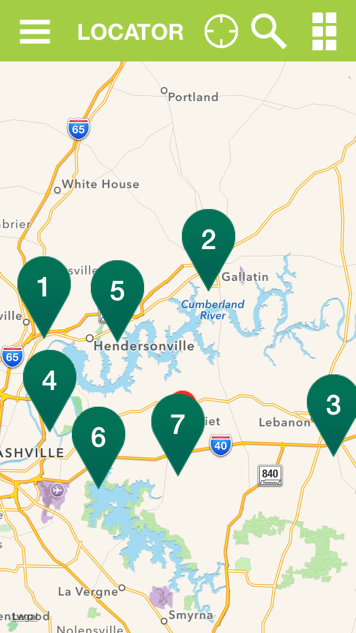
|  |
| --- |
| // Get tracker. Tracker t = ((AnalyticsSampleApp) getActivity().getApplication()).getTracker(TrackerName.APP\_TRACKER); // Build and send an Event. t.send(new HitBuilders.EventBuilder()  .setCategory(getString(categoryId))  .setAction(getString(actionId))  .setLabel(getString(labelId))  .build()); |

# DineTime API Integration

Ignite encapsulates the QSR DineTime API and can provide average wait times for a location that you can display to a user in your custom user interface. Ignite also provides a lightweight API to add, modify and cancel DineTime waitlist bookings. These functionalities are all available via the SDK. The following sections describe how to use the SDK methods to perform these tasks.

NOTE: The examples provided below are in Swift for iOS and Java for Android.

The following diagram illustrates the recommended UI flow as implemented in the SDK Demo application.



use BFWaitlistDP cache methods to determine booking state and display correct UI

### Enhancements to BFConfiguration

In order to take advantage of the Ignite "lite" API that provides the DineTime functionality, ensure that your sdkconfig.json file contains the "igniteLiteURL" key (spelling and capitalization of this key should be exactly as shown here.) The format of the value for this key is as follows (ABCDEFG should be replaced with your Ignite account name). **Do not include a trailing forward slash!**

**Staging:** http://hosted.where2stageit.com/ABCDEFG/lite

**Production:** http://hosted.where2getit.com/ABCDEFG/lite

The BFConfiguration class will read this key and supply it to the SDK functions in the BFWaitlistDP class listed below.

### Displaying DineTime Average Wait Times (Powered by Ignite)

Average wait times are provided for each location returned from ignite (such as via an Ignite **getlist** API call). If your location data contains non-null values for this property in Ignite, you can read that property from an instance of the SDK's **BFLocation** class.

**iOS**

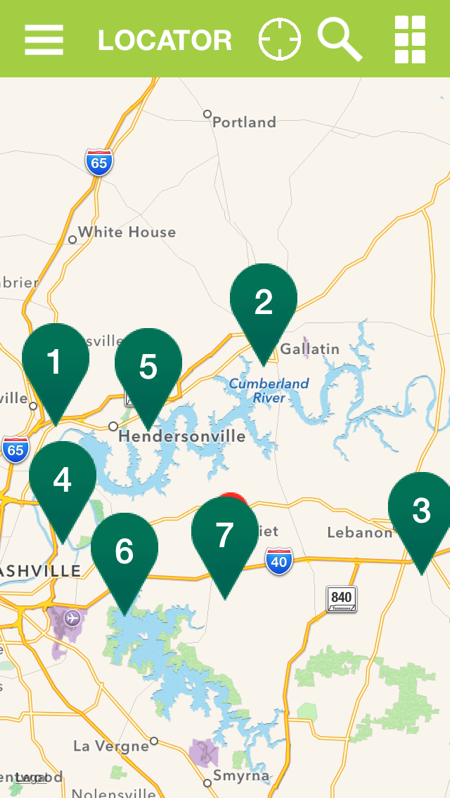
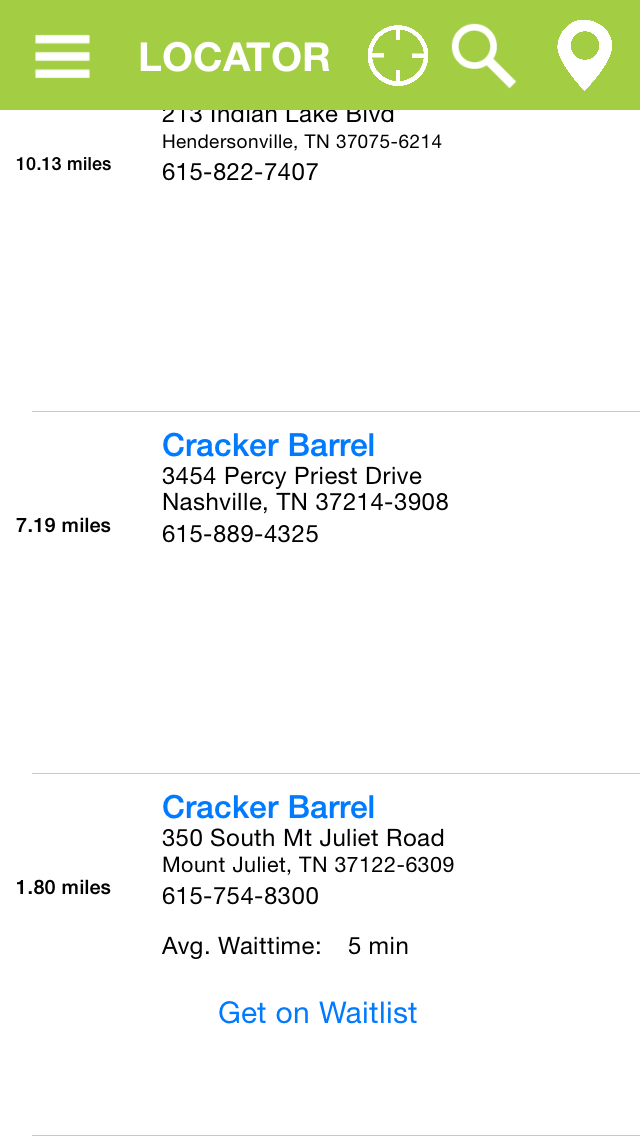
public var avgWaitTime: String?

**Android**

public String avgWaitTime;

For non-null values, display the value of this string. Locations that have null values for this property should be assumed not to support DineTime. The example application will hide all the related UI in the list view if **avgWaitTime** is null or empty.

Note that in the demo app, average wait times and the related UI are only displayed in the list view and not in the map tiles.



### Adding DineTime Waitlist Bookings (Powered by Ignite)

Call the following method on BFWaitlistDP to create a DineTime booking via the Ignite API.

**iOS**

public class func addToDineTimeWaitlist(brandifyAppKey: String,

guestCellNumber: String,

numberOfGuests: Int,

guestFirstName: String,

guestLastName: String? = "",

guestEmailAddress: String,

subscribeToEmails: Bool,

legalTermsAcknowledged: Bool,

clientKey: String,

completion: ((responseJSON: NSDictionary?, error: NSError?) -> ())?)

**Android**

**public void** addToDineTimeWaitlist(

String accountName,  
String brandifyAppKey,  
String guestCellNumber,  
Integer numberOfGuests,  
String guestFirstName,  
String guestLastName,  
String guestEmailAddress,  
Boolean subscribeToEmails,  
Boolean legalTermsAcknowledged,  
String clientKey,  
**final** BFRunnable<BFDineTimeBooking> completion)

This method creates and immediately executes a request to add a user to a DineTime waitlist. Provide a closure for the **completion** argument to handle UI updates upon receiving response from server. **This method DOES NOT save the booking to the waitlist cache.** To cache the booking, call the waitlist cache method described below from the closure you provide to the **completion** argument**.**

The argument **guestLastName** is optional because some implementations may require the user's full name to be stored in the firstName column.

Note: Strips any non-numeric characters and spaces from the cell phone number string. Sending a request with non-numeric characters may result in a server error. A 10-digit phone number is required by the Dine Time API to send SMS confirmations to the user.

### Modifying DineTime Waitlist Bookings (Powered by Ignite)

Call the following method on BFWaitlistDP to cancel a DineTime booking via the Ignite API. The **visitId** parameter is the identifier for the booking to be modified.

Note: The sample app does not do any form validation to determine if there are actual changes in the form, but you may wish to do this to save an unnecessary web service call.

**iOS**

public class func modifyDineTimeWaitlistForVisit(visitId: String,

brandifyAppKey: String,

guestCellNumber: String,

numberOfGuests: Int,

guestFirstName: String,

guestLastName: String? = "",

guestEmailAddress: String,

subscribeToEmails: Bool,

legalTermsAcknowledged: Bool,

clientKey: String,

completion: ((responseJSON: NSDictionary?, error: NSError?) -> ())?)

**Android**

**public void** modifyDineTimeWaitlistForVisit(

String visitId,  
String accountName,  
String brandifyAppKey,  
String guestCellNumber,  
Integer numberOfGuests,  
String guestFirstName,  
String guestLastName,  
String guestEmailAddress,  
Boolean subscribeToEmails,  
Boolean legalTermsAcknowledged,  
String clientKey,  
**final** BFRunnable<BFDineTimeBooking> completion)

This method creates and immediately executes an asynchronous request to modify the booking through the Ignite API. Provide a closure for the **completion** argument to handle UI updates upon receiving response from server. **This method DOES NOT save the booking to the waitlist cache.** To cache the booking, call the waitlist cache method described below from the closure you provide to the **completion** argument**.**

On failure, this method will remove the associated cached booking, if it exists, so that you can provide the user a remedy to create a new booking (e.g. refreshing the UI that starts the flow so that the receive the "Get on Waitlist" button rather than the "Modify Waitlist" button).

Note: Strips any non-numeric characters and spaces from the cell phone number string. Sending a request with non-numeric characters may result in a server error. A 10-digit phone number is required by the Dine Time API to send SMS confirmations to the user.

### Cancelling DineTime Waitlist Bookings (Powered by Ignite)

Call the following method on BFWaitlistDP to cancel a DineTime booking via the Ignite API. The **visitId** parameter is the identifier for the booking to be deleted.

**iOS**

public class func deleteDineTimeWaitlistBooking(

visitId: String,

brandifyAppKey: String,

clientKey: String,

completion: ((responseJSON: NSDictionary?, error: NSError?) -> ())?)

**Android**

**public void** deleteDineTimeWaitlistBooking(

String visitId,

String accountName,

String appKey,

String clientKey,

**final** BFRunnable<BFDineTimeBooking> completion)

This method creates and immediately executes an asynchronous request to delete the booking through the Ignite API. Provide a closure for the **completion** argument to handle UI updates upon receiving response from server. On success, this method will also remove the associated cached booking, if it exists.

## Managing DineTime Waitlist Bookings

The SDK provides a cache to facilitate managing the state of a booking in order to update the UI for a given location based on the state of a booking.

In order to take advantage of this cache system, first call **BFWaitlistDP.initializeWaitlistModule()** to register an entry in NSUserDefaults for use with the waitlist cache methods described in the sections that follow.

### Caching a Booking

When a booking is created via the SDK method described above, the response will include a **visitId**. (In the response JSON, this is the key "ID".) The response is your only opportunity to store this value. The **visitId** is required by the Ignite API to modify or cancel a booking.

To ensure that users only make one booking per location, the cache takes a location identifier (**clientKey**) as the key for each cached booking object.

Call this method on BFWaitlistDP upon successfully adding or modifying a waitlist booking. The key-value pair stored here can later be used to determine if a waitlist entry has been made for a given location.

**iOS**

public class func storeDineTimeVisitForLocation(clientKey: String,

numberOfGuests: Int,

guestFirstName: String,

guestLastName: String? = "",

guestCellNumber: String,

guestEmail: String,

visitId: String)

**Android**

**public void** storeDineTimeVisitForLocation(

String clientKey,

Integer numberOfGuests,

String guestFirstName,

String guestLastName,

String guestCellNumber,

String guestEmail,

String visitId)

Note that **guestLastName** is optional, as some implementations may prefer that the guest's full name is stored in one field.

### Fetching a Booking from the Cache

To modify or cancel an existing booking the booking's **visitId** is required. Retrieve the booking from the cache in order to obtain the **visitId** and the rest of the booking details in order to populate the booking UI to allow the user to modify the details of the booking, then pass the booking info through to the appropriate SDK method to create or modify a booking via the Ignite API.

The following function returns the stored DineTime booking for a given location via the location's Brandify clientKey or nil if not found.

**iOS**

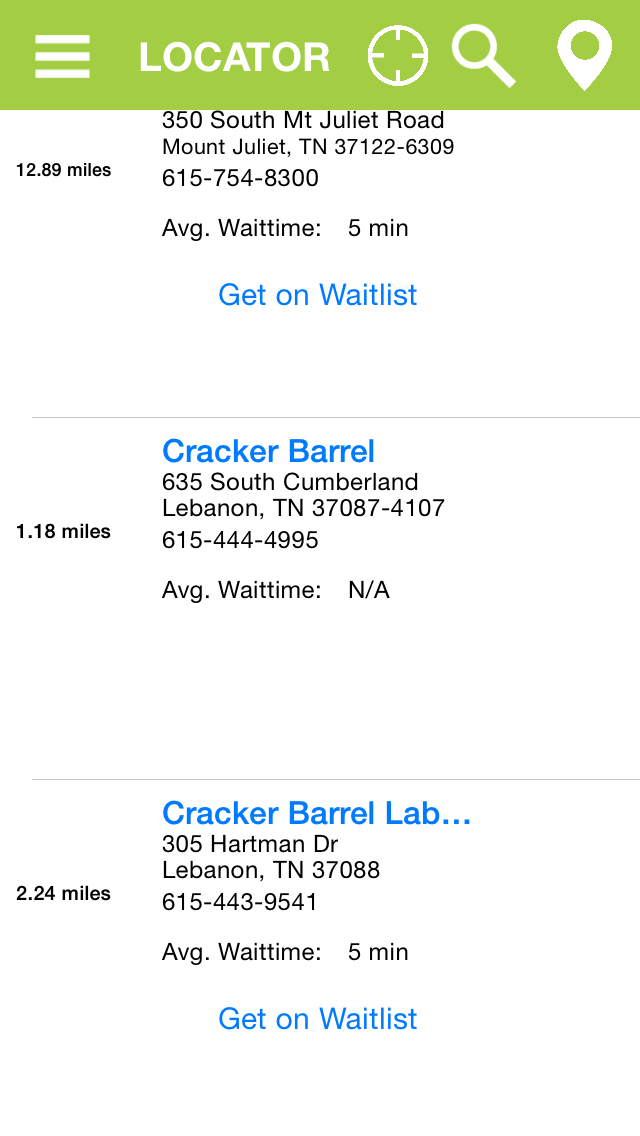
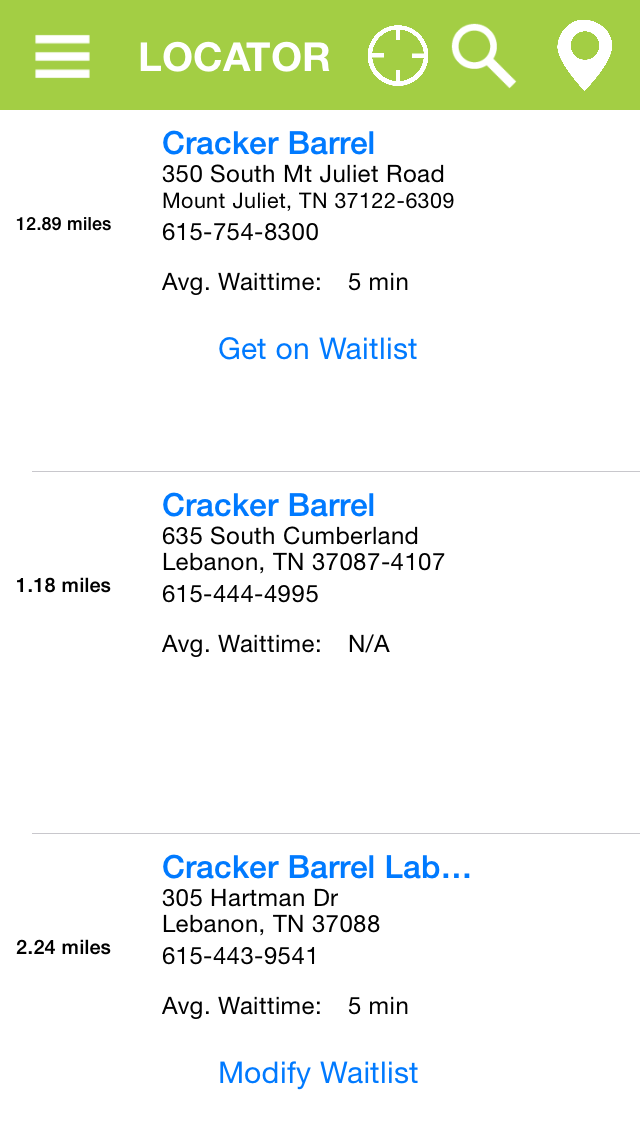
public class func dineTimeBookingForLocation(clientKey: String) -> BFDineTimeBooking?

**Android**

**public** BFDineTimeBooking dineTimeBookingForLocation(String clientKey)

### Removing a Booking from the Cache and Checking for Existing Cached Bookings

When a user decides to cancel a booking or you otherwise decide that the booking is no longer valid, remove it from the cache so that you can return the UI to its original state as illustrated below.



add booking

delete booking

Call the following method on BFWaitlistDP to remove a booking from the cache by location (**clientKey**).

**iOS**

public class func removeVisitForLocationFromCache(clientKey: String)

**Android**

**public void** removeVisitForLocationFromCache(String clientKey)

Once the booking has been removed, refresh your UI to return it to the original state. You can use the following method on BFWaitlistDP to determine if a booking already exists in the cache for a given location.

**iOS**

public class func checkExistingDineTimeWaitlistEntriesForLocation(clientKey: String) -> Bool

**Android**

**public boolean** checkExistingDineTimeWaitlistEntriesForLocation(String clientKey)

Returns true if a booking was found for the given **clientKey**.

### Clearing the Cache

Call the following method on BFWaitlistDP to clear the entire cache. This will also remove the cache's key in NSUserDefaults, so you will need to call BFWaitlistDP.initializeWaitlistModule() before using any of the other waitlist cache methods.

**iOS**

public class func clearDineTimeVisitsCache()

**Android**

**public void** clearDineTimeVisitsCache()

# API Reference

## General Classes

|  |
| --- |
| BFContext Class |

**Comments**

The BFContext class holds a reference to the Brandify AppKey and the Locator Search Rest URL. By default both, the AppKey and the Locator Search Rest URL, are initialized using the values found in the sdkconfig.json file: appkey and locatorSearchRestURL respectively.

In most cases, changing the configuration file to match the app key and locator search URL of your company will be sufficient. If it is necessary to make calls to multiple locator search URLs or to use multiple app keys in the same application, then override the BFContext by initializing by creating a derived class inherited from the BFContext class in your project as necessary.

The following example overrides the BFContext initializer, then uses the derived class to execute a locator search.

|  |
| --- |
| public class MyCompanyContext : BFContext  {  public override init()  {  super.init()  restURL = "<http://hosted.where2getit.com/rest/locatorsearch>"  apiKey = "b82931b1-9874-436a-aefc-97cb3b538e0d"  }  } |

...

|  |
| --- |
| let context = MyCompanyContext()  let locatorDP = BFLocatorDP()  let searchLocation = BFSearchLocation()    locatorDP.search(context, searchLocation: searchLocation) { (returnedLocationList) -> () in  //search results...  }) |

**Properties**

|  |  |
| --- | --- |
| **Name** | **Description** |
| **restURL** | The URL endpoint for rest calls. |
| **appKey** | The unique Brandify app key that scopes to a particular Brandify agency. |

## Model Classes

The following classes represent the SDK data models.

|  |
| --- |
| BFSearchLocation Class |

**Comments**

The BFSearchLocation class is only used as input for searches.

**Properties**

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| **name** | String | The store name of the location. |
| **addressLine** | String | Full or partial address information |
| **country** | String | Scopes the search to a specific country. |
| **longitude** | Float | Latitude and longitude provide the search with specific coordinates to center the search. |
| **latitude** | Float |
| **geoip** | Bool | If true, the client IP address will be used to determine the geo coordinates to center the search. |
| **radiusUOM** | UnitOfMeasure | The Unit of measure by which the radial search will be bounded.  UnitOfMeasure.Miles  UnitOfMeasure.Meters  UnitOfMeasure.Kilometers  Default is Miles. |
| **searchradius** | Array | A list of radial distances. The search will continue to execute in order of each radial distance, from nearest to furthest, until at least one location is found or until all of the radial distances have been searched. |
| **limit** | Int | Maximum locations to return. |
| **filterProperties** | Dictionary<String, String> | Used during the search to filter results. The configuration file contains a ‘where’ key in the JSON file under Formdata. If properties in this dictionary match properties in the configured ‘where’ section, then the values of the ‘where’ properties will be updated with those in the filterProperties. |

|  |
| --- |
| BFLocation Class |

**Comments**

The BFLocation is returned from location searches and holds all information associated with a particular location.

**Properties**

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| **name** | String | The store name of the location. |
| **address1** | String | The street address of the location. |
| **address2** | String | Additional street address information of the location. |
| **city** | String | The city of the location. |
| **state** | String | The state of the location. |
| **postalcode** | String | The postal code of the location. |
| **country** | String | The country of the location. |
| **province** | String | The province of the location. |
| **phone** | String | The phone number of the location. |
| **longitude** | Float | Longitude of location. |
| **latitude** | Float | Latitude of location. |
| **clientKey** | Int | The unique key, with respect to a particular app key, that identifies a single store. |
| **icon** | String | Key value that maps to the configuration file ‘icons’ section to the location. Use this value along with the ‘icons’ array in the configuration file to determine the image URL to use for the map pin. |

The properties listed above are always located on the BFLocation class. However, most clients will have many more properties associated with a location than those properties listed. These properties, custom properties, should be retrieved through the getCustomProperty() method noted below.

**Methods**

**getCustomProperty(propertyName)**

*Returns*

AnyObject? – The value of the property retrieved.

*Method Description*

Retrieves a custom property based on the property name.

*Method Parameters*

**propertyName : String** – The name of the custom property to retrieve.

|  |
| --- |
| BFListOfLocations Class |

**Comments**

The BFListOfLocations class is returned in a location search and contains BFLocation instances for all matching locations.

**Properties**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| **responseCode** | Int | The code value ‘1’ indicates that results have been returned. All other codes indicate that results are not available. |
| **locations** | Array<BFLocation> | Collection of location results. |
| **radius** | Float | Radius of the search area. |
| **radiusUOM** | String | The radius unit of measure. |
| **longitude** | Float | Longitude of the center of the search. |
| **latitude** | Float | Latitude of the center of the search. |

|  |
| --- |
| BFConfiguration Class |

**Comments**

The BFConfiguration class holds Brandify Mobile SDK configuration information.

**Properties**

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| **appKey** | String | Unique app identifier. |
| **locatorSearchRestURL** | String | REST URL path for locator searches. |
| **netconfig** | String | URL path to the configuration file on the net. |
| **locator** | LocatorConfig | Locator configuration information. |
| **icons** | Dictionary<String, String> | Dictionary of image URLs. The key to the value of this dictionary will map to the location ***icon*** property. |
| **logos** | Dictionary <String, String> | Dictionary of image URLs. |
| **defaultStoreImagePathURL** | String | URL path to the default store image. |

|  |
| --- |
| LocatorConfig Class |

**Comments**

The LocatorConfig class contains the formdata configuration.

**Properties**

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| **formdata** | FormdataConfig | Config class container for formdata configuration. |

|  |
| --- |
| FormdataConfig Class |

**Comments**

The FormdataConfig class contains default configuration information that drives locator searches.

**Properties**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Description** | **Default** |
| **geoip** | Bool | Specifies whether the locator search will use the client IP address to center the search. | true |
| **limit** | Int | The maximum number of results that can be returned in one request. | 250 |
| **geolocs** | GeoLocsConfig | Container class for requesting multiple search coordinates. |  |
| **searchRadius** | String | Search radius values. A single numerical value or an array of values separated by the | character. This configuration value causes the search to execute in order of each radial distance, from nearest to furthest, until at least one location is found or until all of the radial distances have been searched. | "10|25|50|100|250" |
| **order** | String | Specifies how the locator search results will be ordered. | NAME, \_DISTANCE DESC |
| **radiusUOM** | String | Specifies the unit of measure for the searchRadius value.  Possible values:  miles  meters  kilometers | Miles |

|  |
| --- |
| GeolocConfig Class |

**Comments**

The GeolocConfig class contains default configuration information that drive locator searches by specific geolocation properties.

**Properties**

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| **addressLine** | String | Full or partial address information |
| **country** | String | Scopes the search to a specific country. |
| **latitude** | Float | Latitude and longitude provide the search with specific coordinates to center the search. |
| **longitude** | Float |

|  |
| --- |
| GeolocsConfig Class |

**Comments**

The GeolocsConfig class is a container class for lists of GeolocConfig class instances.

**Properties**

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| **geolocs** | GeolocConfig Array | A container for lists of GeolocConfig class instances. |

## Data Provider Classes

The following classes represent the classes that broker sending and receiving data.

|  |
| --- |
| BFMobileSDK Class |

**Comments**

The purpose of the BFMobileSDK is to explicitly initialize key aspects of the SDK at the right code location for each implementation.

**Methods**

**iOS**

**initialize(useLocalConfigs, trackLocationInBackground)**

**Android**

**initialize(context, useLocalConfigs, trackLocationInBackground)**

*Returns*

Void

*Method Description*

Initializes the SDK. During this call the configuration is gathered locally and over the network (unless useLocalConfigs is set to false). Location management is also initialized at this time.

*Method Parameters*

**context:Context** (Android Only) – Provides an activity context in order for the SDK to conduct specific Android operations that require it. For instance, the Brandify Mobile SDK uses the Android API to prompt for user permissions. The Android API requires an activity context to execute the API calls involved in that operation.

**useLocalConfigs : Bool** – If set to true, only the local sdkconfig.json file will be used to initialize the configuration. If set to false, the local sdkconfig.json will be used to establish the configuration and then an attempt to retrieve the sdkconfig.json over the network will be executed. If the network sdkconfig.json is found, the property values in this configuration will overwrite any corresponding local configuration property values.

**trackLocationInBackground : Bool** – If set to true, then location information may be collected while the application is in the background. If set to false, then location information will not be collected in the foreground, buy may collect information while the application is visible (in the foreground).

The following classes represent the classes that broker sending and receiving data.

|  |
| --- |
| BFLocatorDP Class |

**Comments**

The BFLocatorDP class is the data provider for locator searches.

**Methods**

**search(context, searchLocation, completionBlock)**

*Returns*

Void

*Method Description*

Executes locator searches using the BFSearchLocation model class to drive the searches.

*Method Parameters*

**context : BFContext** - Provides the context for the execution of the search. Currently, the context contains connection related information (app key and the REST URL for locator searches). In future releases, the context may also contain security related information.

**searchLocation : BFSearchLocation** - The search location parameter carries information about how to conduct the search. This includes location address, request IP address, or coordinate information about where to center the search. Other filter information such as search radius and property conditions help to bound the search.

**completionBlock : (returnListOfLocations : BFListOfLocations) -> ())** - The completion block is called after the asynchronous search operation has completed. The block returns with the list of locations. The list of locations is always initialized, but may contain a count of zero. If the search was successful then the Status property on the returnListOfLocations instance will equal zero (OK), otherwise the search was not successful and the Description property on the retu will contain more information. It the search encountered an exception on the server, the StackTrace property on the list of locations class will be non-null.

|  |
| --- |
| BFConfigurationDP Class |

**Comments**

The BFConfigurationDP class is the data provider for retrieving Brandify Mobile SDK configuration information. Note that locator searches internally use the BFConfigurationDP to retrieve search default values. For instance, the limit for search results is determined from the configuration file. When executing a locator search, these defaults can be overridden by programmatically setting the values on the BFSearchLocation instance class that drives the search.

**Methods**

**getConfiguration(forceRefresh)**

*Returns*

BFConfiguration instance

*Method Description*

Returns an instance of the BFConfiguration class. The retrieval of the configuration follows these steps.

* If forceRefresh is true, then clear the existing cached configuration.
  + Load the configuration file, sdkconfig.json, from the main application bundle into cache.
  + Inspect the local sdkconfig.json for a 'netconfig' value.
  + If 'netconfig' value exists, then make a network request for that file.
  + If the network request returns a valid JSON file, overwrite any matching configuration properties in the local cached configuration (keeping any non-matching properties as they were.)
* If forceRefresh is false, then load configuration from local cache.

*Method Parameters*

forceRefresh - Boolean value which determines if the current cached configuration file should be refreshed or to use the existing cached version.