support@manago.ai
+48 533 600 808

Integrate Manago AI with your Mobile App

REACT Native

Contents

1. Prerequisites

When integrating your mobile app with Manago AI, bear in mind the following requirements:

- **Minimum supported versions:**
    - Android: min. 7.0 Nougat (API level 24)
    - iOS: min. 15.0 (iPhone versions 6s and above)
    - React Native: min. 0.68.0
- **Supported React Native architectures:**
    - Old architecture: YES
    - New architecture (Fabric): YES
- **Permissions:** To be able to send notifications to your app users, you need two types of permissions, both of which must be requested by your app:
    - System permission (all iOS versions, for Android – versions 13 [API level 33] and above)
    - Marketing consent

  The status of these two permissions must be transferred to Manago AI (for more details, see Section 4 below).

NOTE: *Currently, there is **no option to reassign** a device to a different Contact (different email address).*

EXAMPLE: *After using a development version of your mobile app, you want to test notifications for a different user. The execution of the* `updateContactProperties` *or* `updateContactData` *methods with a different email address will not assign the device to the new email address.*

2. SDK installation and initialization

2.A. Installation

To install the `@salesmanago/mobile-push-sdk-react-native` library, use NPM or Yarn:

```Shell
npm install @salesmanago/mobile-push-sdk-react-native
```

```Shell
yarn add @salesmanago/mobile-push-sdk-react-native
```

To make Manago AI mobile packages available in your project, add the following configuration to your project-level Gradle settings.

Open your `settings.gradle` or `build.gradle` file (at the `rootProject` level) and add the following section inside the `repositories` block:

```Groovy
rootProject.allprojects {
    repositories {
        maven {
            url = uri("https://europe-central2-maven.pkg.dev/salesmanago-mvn/mobile")
        }
    }
}
```

Make sure this configuration is placed in the `repositories` section of the main project, so Gradle can correctly resolve and download dependencies from Manago AI's private Maven repository.

2.B. Initialization

To initialize the Manago AI SDK, call the `init()` method, which passes the `applicationContext` object and Manago AI API key. This method must be called at the app's startup.

The required API key can be generated in Manago AI (**Menu > Channels > Mobile Push > Settings > Integration settings > *API keys* tab**).

```TypeScript
import { Salesmanago } from '@salesmanago/mobile-push-sdk-react-native';

const sdk = new Salesmanago();
sdk.init(<API key>)
```

2.C. Initialization listener and SDK readiness

You can also pass InitializationListener to receive initialization completion callbacks.

`isSuccess` means SDK core initialization is complete and command queue processing can continue. Push token synchronization is performed asynchronously from FCM callbacks, with a fallback token fetch during initialization.

```
TypeScript
const sdk = new Salesmanago();
const result = await sdk.initWithResult('<API key>');
if (result.isSuccess) {
  // SDK initialized successfully
} else {
  // Initialization failed
  // result.errorMessage contains details when available
}
```

`initWithResult()` resolves to:

- `isSuccess: boolean`
- `errorMessage: string | null`

When apiKey is missing or blank, the method resolves with:

```
TypeScript
{ isSuccess: false, errorMessage: 'Missing apiKey argument' }
```

2.D. Reinitialization during runtime

The SDK supports runtime reinitialization with a new API key. This allows switching between different Manago AI API keys during the application's lifecycle without restarting the app. This lets you change the Manago AI account to which the data is sent, which is especially useful for apps supporting multiple regions.

The SDK switches to the new API key, reinitializes all necessary components, and performs an initial synchronization.

The SDK keeps track of existing Mobile App User IDs and tries to re-use the same IDs when switching back to a previously used API key.

To reinitialize the SDK with a new API key, call the `reInit(apiKey)` method:

```
TypeScript
sdk.reInit(<new API key>)
```

Important:
Switching the API key does not automatically opt out Contacts from Mobile Push notifications. If you want to send Mobile Push notifications only from the Manago AI account associated with the new API key, consider revoking the consent for marketing notifications first, before executing the API key swap. This way, after changing the API key, Contacts will no longer receive marketing Mobile Push notifications; however, the transactional ones, such as order confirmations, will still be delivered.

3. Mobile Push and In-App setup

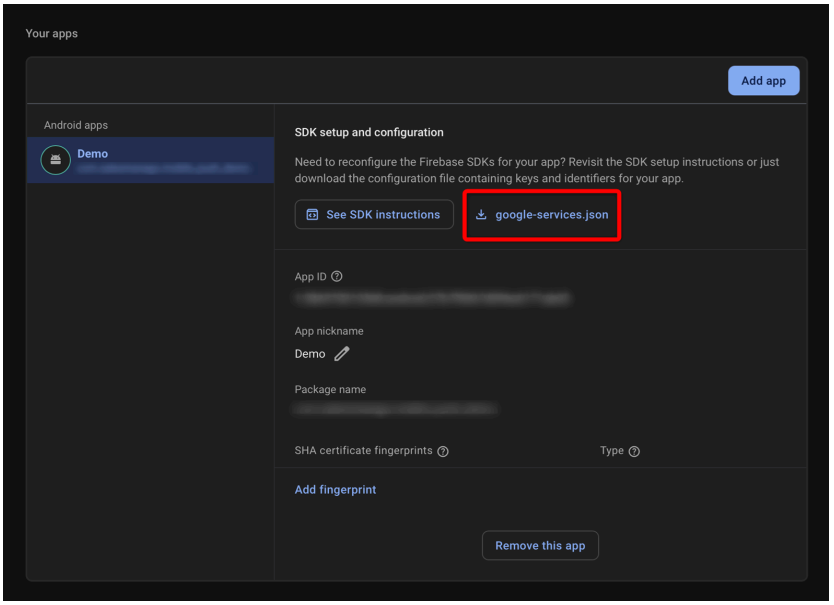
Push notifications are sent via Firebase Cloud Messaging on Android and the Apple Push Notification service (APNs) on iOS. To ensure that your notifications function as intended, perform the configuration steps described below.

3.A. Android: Connecting Manago AI to Firebase

To integrate Manago AI with your mobile app, include your google-services.json file (required to initialize the Manago AI SDK) in the app-level root directory.

To obtain the google-services.json file:

1. Open the Firebase Console and select your project.
2. Click the gear icon next to *Project Overview* and select *Project Settings*.
3. Scroll down to the *Your apps* section, select your Android app, and click the download button for the google-services.json file.



Note that you can also obtain your Android app's config file via REST API (**projects.androidApps.getConfig >>**).

Place the downloaded JSON file in the app-level directory of your app. Make sure that you only have the most recently downloaded config file in your app.

At this stage, add the Gradle plugin named “Google Services” (for example, from the **MVN Repository >>**) to your **project-level** `build.gradle` file:

```groovy
plugins {
    // ...

    // dependency for the Google services Gradle plugin
    id("com.google.gms.google-services") version "<version>" apply false
}
```

and to the **app-level** `build.gradle` file:

```groovy
plugins {
    id("com.android.application")

    // Google services Gradle plugin
    id("com.google.gms.google-services")

    ...
}
```

Push notifications on Android are sent via Firebase Cloud Messaging. The SDK handles FCM token management automatically. The app only needs Firebase setup and notification permissions for the default flow.

Android default SDK flow

By default, no extra service setup is required in the app. The native Manago AI Android SDK registers its internal FirebaseMessagingService and handles SDK payloads automatically.

Dedicated notification icon

If you want to use a dedicated notification icon for Android push notifications instead of the application icon, add this meta-data entry inside the application tag in `android/app/src/main/AndroidManifest.xml`.

```xml
<application ...>
    <meta-data android:name="com.salesmanago.mobilepush.notification_small_icon"
android:resource="@drawable/ic_stat_salesmanago" />
</application>
```

If this value is not provided, the SDK checks `com.google.firebase.messaging.default_notification_icon` first and then falls back to the application icon. Use a monochrome small icon for best status bar visibility.

Android RN-provided service

If you need the React Native package to render Manago AI PUSH payloads itself while still delegating SDK IN_APP/COMBO handling, this package provides `com.mobilepushsdkreactnative.messaging.SalesmanagoMessagingService`.

To enable it, replace the SDK internal messaging service in `AndroidManifest.xml`:

```xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:tools="http://schemas.android.com/tools">
    <application ...>
        <service android:name="com.salesmanago.library.common.messaging.MessagingService"
tools:node="remove" />
        <service
android:name="com.mobilepushsdkreactnative.messaging.SalesmanagoMessagingService"
android:exported="false" tools:node="replace">
            <intent-filter>
                <action android:name="com.google.firebase.MESSAGING_EVENT" />
            </intent-filter>
        </service>
    </application>
</manifest>
```

Notes:

- Non-Manago AI payloads are ignored by this provided service.
- Manago AI IN_APP and COMBO payloads are delegated to the SDK internal handler.
- Manago AI PUSH payloads are rendered by the React Native package path with hardened image loading: connectTimeout/readTimeout = 5000ms, non-2xx errorStream drain.
- Keep only one MESSAGING_EVENT service active: SDK default service, RN-provided service, or app custom service.

Android app-owned custom service

If you need custom behavior for non-Manago AI pushes, create your own FirebaseMessagingService. Split payloads yourself and forward only Manago AI payloads via SalesmanagoPushBridge.

`SalesmanagoPushBridge.handleSalesmanagoMessage(...)` forwards Manago AI payloads to the native Android SDK handler. It is intended for apps that own their FCM service and need to keep non-Manago AI payloads in app-specific flow.

```kotlin
import com.google.firebase.messaging.FirebaseMessagingService
import com.google.firebase.messaging.RemoteMessage
import com.mobilepushsdkreactnative.messaging.SalesmanagoPushBridge

class CustomAppMessagingService : FirebaseMessagingService() {
    override fun onMessageReceived(message: RemoteMessage) {
        if (SalesmanagoPushBridge.isSalesmanagoPayload(message.data)) {
            SalesmanagoPushBridge.handleSalesmanagoMessage(this, message)
        } else {
            // Handle non-Manago AI payload in your own flow.
        }
    }

    override fun onNewToken(token: String) {
        SalesmanagoPushBridge.handleSalesmanagoToken(this, token)
    }
}
```

Important notes:

- Keep only one MESSAGING_EVENT service active: SDK default service, RN-provided service, or app custom service.
- If you need full custom rendering for non-Manago AI notifications in background, prefer data-only FCM payloads without top-level notification.

3.B. Android: Handling In-App notifications

To make sure In-App notifications are displayed correctly in your React Native app on Android, you need to properly configure your main activity.

Check your `AndroidManifest.xml` file and look at the `<activity>` tag for your main activity. Make sure the `android:taskAffinity` attribute is not set to an empty string. If `taskAffinity` is empty, Android may block notifications from appearing properly.

Correct:

```xml
<activity
    android:name=".MainActivity"
    android:taskAffinity="com.example.myapp" />
```

Incorrect:

```xml
<activity
    android:name=".MainActivity"
    android:taskAffinity="" />
```

If you do not set `android:taskAffinity` at all, Android will automatically assign a default value based on your app's package name, which works correctly for displaying In-App notifications.

3.C. iOS: Handling In-App notifications

To enable the handling of In-App notifications by your mobile app, implement the relevant AppDelegate methods.

Swift

```
import React
import SalesmanagoSDK
import UserNotifications

@main
class AppDelegate: RCTAppDelegate, UNUserNotificationCenterDelegate {
  private var salesmanagoInitObserver: NSObjectProtocol?

  override func application(
    _ application: UIApplication,
    didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOptionsKey: Any]? =
nil
  ) -> Bool {
    self.moduleName = "YourAppName"
    self.initialProps = [:]

    // Deferred until the SDK finishes initializing: calling this before
    // Salesmanago.initialize(...) completes drops cold-start push-click tracking.
    salesmanagoInitObserver = NotificationCenter.default.addObserver(
      forName: Notification.Name("MobilePushSdkReactNativeDidInitialize"),
      object: nil,
      queue: .main
    ) { [weak self] notification in
      guard (notification.userInfo?["isSuccess"] as? Bool) == true else { return }
      if let observer = self?.salesmanagoInitObserver {
        NotificationCenter.default.removeObserver(observer)
        self?.salesmanagoInitObserver = nil
      }
      Salesmanago.application(didFinishLaunchingWithOptions: launchOptions)
    }

    Salesmanago.setNotificationDelegate(self) // required for non-Manago AI alert/tap
callbacks
    return super.application(application, didFinishLaunchingWithOptions: launchOptions)
  }

  override func application(
    _ application: UIApplication,
    didRegisterForRemoteNotificationsWithDeviceToken deviceToken: Data
  ) {
    Salesmanago.sendDeviceTokenData(deviceToken)
  }
```

```
  override func application(
    _ application: UIApplication,
    didReceiveRemoteNotification userInfo: [AnyHashable: Any],
    fetchCompletionHandler completionHandler: @escaping (UIBackgroundFetchResult) -> Void
  ) {
    if Salesmanago.isSdkNotification(userInfo: userInfo) {
      Salesmanago.didReceiveRemoteNotification(userInfo: userInfo)
      completionHandler(.noData)
      return
    }
    // Handle non-Manago AI payload in your own push flow.
    // Make sure completionHandler is called exactly once on this path.
    completionHandler(.noData)
  }

  override func userNotificationCenter(
    _ center: UNUserNotificationCenter,
    willPresent notification: UNNotification,
    withCompletionHandler completionHandler: @escaping (UNNotificationPresentationOptions)
-> Void
  ) {
    let userInfo = notification.request.content.userInfo
    if Salesmanago.isSdkNotification(userInfo: userInfo) {
      completionHandler([])
      return
    }
    // Handle non-Manago AI push in your own flow.
    completionHandler([.banner, .list, .sound])
  }

  override func userNotificationCenter(
    _ center: UNUserNotificationCenter,
    didReceive response: UNNotificationResponse,
    withCompletionHandler completionHandler: @escaping () -> Void
  ) {
    let userInfo = response.notification.request.content.userInfo
    if !Salesmanago.isSdkNotification(userInfo: userInfo) {
      // Handle non-Manago AI tap in your own flow.
    }
    completionHandler()
  }
}
```

C/C++

```
#import "AppDelegate.h"
```

```
#import <React/RCTBundleURLProvider.h>
#import <SalesmanagoSDK/SalesmanagoSDK-Swift.h>
#import <UserNotifications/UserNotifications.h>


#import "AppDelegate.h"
#import <React/RCTBundleURLProvider.h>
#import <SalesmanagoSDK/SalesmanagoSDK-Swift.h>
#import <UserNotifications/UserNotifications.h>

@interface AppDelegate () <UNUserNotificationCenterDelegate>
@end

@implementation AppDelegate

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary
*)launchOptions {
  self.moduleName = @"YourAppName";
  self.initialProps = @{};

  // Deferred until the SDK finishes initializing: calling this before
  // Salesmanago.initialize(...) completes drops cold-start push-click tracking.
  __block __weak id weakObserver = nil;
  id observer = [[NSNotificationCenter defaultCenter]
addObserverForName:@"MobilePushSdkReactNativeDidInitialize" object:nil
queue:[NSOperationQueue mainQueue] usingBlock:^(NSNotification *notification) {
    NSNumber *isSuccess = notification.userInfo[@"isSuccess"];
    if (isSuccess.boolValue) {
      id strongObserver = weakObserver;
      if (strongObserver) {
        [[NSNotificationCenter defaultCenter] removeObserver:strongObserver];
      }
      [Salesmanago applicationWithDidFinishLaunchingWithOptions:launchOptions];
    }
  }];
  weakObserver = observer;

  [Salesmanago setNotificationDelegate:self]; // required for non-Manago AI alert/tap
callbacks
  return [super application:application didFinishLaunchingWithOptions:launchOptions];
}

- (void)application:(UIApplication *)application
didRegisterForRemoteNotificationsWithDeviceToken:(NSData *)deviceTokenData {
  [Salesmanago sendDeviceTokenData:deviceTokenData];
}

- (void)application:(UIApplication *)application didReceiveRemoteNotification:(NSDictionary
*)userInfo fetchCompletionHandler:(void (^)(UIBackgroundFetchResult
result))completionHandler {
```

```
  if ([Salesmanago isSdkNotificationWithUserInfo:userInfo]) {
    [Salesmanago didReceiveRemoteNotificationWithUserInfo:userInfo];
    completionHandler(UIBackgroundFetchResultNoData);
    return;
  }
  // Handle non-Manago AI payload in your own push flow.
  // Make sure completionHandler is called exactly once on this path.
  completionHandler(UIBackgroundFetchResultNoData);
}

- (void)userNotificationCenter:(UNUserNotificationCenter *)center
willPresentNotification:(UNNotification *)notification withCompletionHandler:(void
(^)(UNNotificationPresentationOptions options))completionHandler {
  NSDictionary *userInfo = notification.request.content.userInfo;
  if ([Salesmanago isSdkNotificationWithUserInfo:userInfo]) {
    completionHandler(0);
    return;
  }
  // Handle non-Manago AI push in your own flow.
  completionHandler(UNNotificationPresentationOptionBanner |
UNNotificationPresentationOptionList | UNNotificationPresentationOptionSound);
}

- (void)userNotificationCenter:(UNUserNotificationCenter *)center
didReceiveNotificationResponse:(UNNotificationResponse *)response
withCompletionHandler:(void (^)(void))completionHandler {
  NSDictionary *userInfo = response.notification.request.content.userInfo;
  if (![Salesmanago isSdkNotificationWithUserInfo:userInfo]) {
    // Handle non-Manago AI tap in your own flow.
  }
  completionHandler();
}

@end
```

Multi-provider Push Setup

When integrating with multiple push providers, follow these guidelines:

- Route only Manago AI payloads to Salesmanago.didReceiveRemoteNotificationWithUserInfo.
- Keep non-Manago AI handling in your own flow.
- Call Salesmanago.setNotificationDelegate(...) so non-Manago AI alert/tap callbacks are forwarded.
- Make sure completionHandler is called exactly once per notification.
- Salesmanago.isSdkNotificationWithUserInfo(...) classifies SDK payloads by custom.sender == "salesmanago" case-insensitively.

Device token

If you need to manually send a device token to the SDK, for example when using a custom push notification provider, use `sendDeviceToken()`.

```
TypeScript
Salesmanago.sendDeviceToken("<device-token>");
```

On iOS, this forwards the APNs token to the native SDK. On Android, FCM token synchronization is handled by the native SDK messaging integration; manual `sendDeviceToken()` is currently not required for the standard Android flow.

3.D. Rich Push Notifications on iOS

To display images and other media in push notifications on iOS, your application requires a **Notification Service Extension**. This extension allows the app to process the notification payload and download the media before the notification is presented to the user.

Manual Setup Steps in Xcode:

1. Open your project in Xcode and go to **File > New > Target...**
2. Select **Notification Service Extension** and click Next.
3. Enter a name for the extension (e.g., `NotificationService`) and ensure the project and embedding app are correctly selected.
4. **Add SalesmanagoSDK Framework to the Extension Target:**
    a. Select your project in the Project Navigator
    b. Select the Notification Service Extension target
    c. Go to the General tab
    d. Under Frameworks and Libraries, click +
    e. Add SalesmanagoSDK.xcframework
    f. Make sure the framework is set to Embed & Sign
5. Replace the contents of the generated `NotificationService.swift` file with:

Swift

```
import SalesmanagoSDK
import UserNotifications

class NotificationService: UNNotificationServiceExtension {
    var contentHandler: ((UNNotificationContent) -> Void)?
    var bestAttemptContent: UNMutableNotificationContent?

    override func didReceive(
        _ request: UNNotificationRequest,
        withContentHandler contentHandler: @escaping (UNNotificationContent) -> Void
    ) {
        self.contentHandler = contentHandler
        bestAttemptContent = (request.content.mutableCopy() as?
UNMutableNotificationContent)

        guard let content = bestAttemptContent else {
            contentHandler(request.content)
            return
        }

        Salesmanago.didReceive(request)

        guard let imageURL = URL(string: content.launchImageName) else {
            contentHandler(content)
            return
        }

        downloadImage(from: imageURL) { attachment in
            if let attachment {
                content.attachments = [attachment]
            }
            contentHandler(content)
        }
    }

    override func serviceExtensionTimeWillExpire() {
        if let contentHandler, let bestAttemptContent {
            contentHandler(bestAttemptContent)
        }
    }
}

private extension NotificationService {
    func downloadImage(from url: URL, completion: @escaping (UNNotificationAttachment?) ->
Void) {
        let task = URLSession.shared.dataTask(with: url) { data, _, error in
            guard let data, error == nil else {
                completion(nil)
                return
            }
```

```
            do {
                let tempDirectory = FileManager.default.temporaryDirectory
                let uniqueFilename = UUID().uuidString + "-" + url.lastPathComponent
                let fileURL = tempDirectory.appendingPathComponent(uniqueFilename)
                try data.write(to: fileURL)
                let attachment = try UNNotificationAttachment(
                    identifier: "imageAttachment",
                    url: fileURL,
                    options: nil
                )
                completion(attachment)
            } catch {
                completion(nil)
            }
        }
        task.resume()
    }
}
```

Note: the image URL is read from `content.launchImageName`, not from a custom userInfo key. The SDK's push payload places the image URL there so it survives the standard UNNotificationContent mutable copy without extra parsing.

Important Payload Requirement:
For the system to trigger your extension, the push notification payload sent from Manago AI must include the `"mutable-content": 1` flag. Without this key, the notification will be displayed as a standard text-only alert even if the extension is correctly configured.

3.E. In-App Queue Interval

The SDK enforces a minimum delay between consecutive in-app message displays (default: 60 seconds). You can configure this interval at runtime using the configure() method. The value is specified in seconds.

Setting the interval to 0 disables the delay, allowing in-app messages to display back-to-back. If a shorter interval is set while a display is already scheduled, and the new interval has already elapsed, the next in-app will be shown immediately.

TypeScript

```
sdk.configureInAppQueueInterval(10);
```

3.F. Deep links

Just like standard links lead to webpages, **deep links** redirect users to specific locations within your application (for example, a product view).

If you want to use deep links in Mobile Push and/or In-App notifications, you need to perform additional configuration for both Android and iOS.

Android

For Android, each deep link needs to be declared in your app.

The `AndroidManifest.xml` file declares your main activity (usually `MainActivity`) within the `<activity>` tag. To enable this activity to handle deep links, declare your `scheme` in a separate `<intent-filter>`. For instance, if you create a notification in Manago AI with the following deep link: `salesmanago://main`, the `<activity>` tag should contain the following declarations:

```
XML
<activity android:name=".MainActivity" ...>
  <intent-filter>
      ...
  </intent-filter>
  <intent-filter>
      <action android:name="android.intent.action.VIEW" />
      <category android:name="android.intent.category.DEFAULT" />
      <category android:name="android.intent.category.BROWSABLE" />
      <data android:scheme="salesmanago" />
  </intent-filter>
</activity>
```

Replace 'salesmanago' in the example above with whatever URL scheme you configured in the SALESmanago dashboard.

iOS

For iOS, the deep link scheme has to be declared in your `Info.plist` file. For instance, if you create a notification in Manago AI with the following deep link: `salesmanago://main`, your `Info.plist` file should contain the following declarations:

```
XML
<key>CFBundleURLTypes</key>
<array>
    <dict>
        <key>CFBundleTypeRole</key>
        <string>Editor</string>
        <key>CFBundleURLName</key>
        <string>YOUR APP BUNDLE IDENTIFIER</string>
        <key>CFBundleURLSchemes</key>
        <array>
            <string>salesmanago</string>
        </array>
    </dict>
</array>
```

To handle the incoming URL in your `AppDelegate`, add the following code:

Swift

```swift
import React

override func application(
  _ app: UIApplication,
  open url: URL,
  options: [UIApplication.OpenURLOptionsKey: Any] = [:]
) -> Bool {
  return RCTLinkingManager.application(app, open: url, options: options)
}
```

C/C++

```
#import <React/RCTLinkingManager.h>

- (BOOL)application:(UIApplication *)application openURL:(NSURL *)url
options:(NSDictionary<UIApplicationOpenURLOptionsKey,id> *)options {

  return [RCTLinkingManager application:application openURL:url options:options];

}
```

You can also register one or more URL schemes in Xcode.

Read more in Apple's documentation >>

URL scheme registration

1. Open the *Info* tab of your project settings.
2. In the *URL Types* section, declare all URL schemes supported by your app. Click the plus button and provide the required details.
    - The **identifier** is your mobile app's bundle ID (also used when integrating Manago AI with iOS).
    - The **URL scheme** must be unique. To ensure that your URL scheme is unique, we recommend using your brand or application name in it, for example: *benhauer-salesmanago*.

3.G. Web links

By default, HTTP(S) URLs contained in a push notification are opened in the device's external browser. Applications can instead configure the SDK to open these links in an in-app browser based on Android Custom Tabs/SFSafariViewController, giving a more integrated experience without leaving the app.

TypeScript

```typescript
import {
  Salesmanago,
```

```
  UrlPresentation,
} from '@salesmanago/mobile-push-sdk-react-native';

const sdk = new Salesmanago();
sdk.configureUrlPresentation(UrlPresentation.IN_APP_BROWSER);
```

Affects only HTTP(S) URLs — deep links and other custom schemes keep using the system URL handler.

3.H. Cold-start reliability

Cold-start reliability: a push notification tapped while the app was killed reaches the app as two separate intents: a plain launch intent, followed by the deep link intent itself. On Android, the second one can arrive before your JS code has attached its own Linking listeners and get silently dropped. Call getInitialDeepLink() once after your first render, in addition to Linking.getInitialURL(), to recover it independently of Linking:

TypeScript

```
const sdk = new Salesmanago();
useEffect(() => {
  sdk.getInitialDeepLink().then((url) => {
    if (url) {
      // route to url, e.g. via your navigation library's linking config
    }
  });
}, []);
```

This resolves to null on iOS, where Linking/RCTLinkingManager already handle cold-start deep links reliably.

Replace 'salesmanago' in the example above with whatever URL scheme you configured in the SALESmanago dashboard.

4. Managing consents

Manago AI expects your application to transfer two distinct permissions for Push notifications: system permission and marketing consent. This configuration must be performed within the mobile app's code.

NOTE: *The order in which these permissions are requested is not important. However, users who have already granted marketing consent may be more likely to give system permission as well.*

System permission

Your application must request system permission to display notifications. The moment the permission request is shown to the user can be configured in the app's code.

When the system permission is granted or denied, this status must be explicitly transferred to Manago AI using the `Salesmanago.updatePushNotificationSystemPermissions();` method.

If a user **dismisses** the permission request, no information is transferred to Manago AI or Android and the request **can** be shown to this user again.

On Android, call `updatePushNotificationSystemPermissions(...)` after permission changes or when your app returns to foreground. The Android native SDK reads the current system permission state internally, so the granted argument used by the cross-platform wrapper is ignored on Android.

If a user **denies** the permission request, it **cannot** be shown to this user again.

If you attempt to obtain the permission by opening the device's notification settings for your app, once the user returns to the application, use the `Salesmanago.updatePushNotificationSystemPermissions()` method to update the permission status.

Additionally, every time you call the `init` method in Android, the permission status is automatically updated in Manago AI (for example, when a user blocks notifications on the list of notifications or changes the permission status in Android's notification settings).

In iOS, the `init` method does not update the system permission status.

Marketing consent

In addition to system permission, Manago AI requires obtaining marketing consent for displaying Mobile Push and In-App notifications. The way this consent is acquired (for example, its format, appearance, and display time) is fully configurable on your side.

The marketing consent request can be shown to the same user multiple times. Its status should be transferred to Manago AI for both Contacts and anonymous app users, using the `Salesmanago.updateMobilePushOptIn(OptInOption)` method. An example is provided below, and the **possible statuses are described in Section 5**.

NOTE: *If you plan to display the marketing consent as a pop-up and want to prevent it from reappearing after consent has been given, ensure that the information about its acceptance (consent status:* `GRANTED`*) is stored by your mobile app. This must be configured on your side.*

```
TypeScript
sdk.updateMobilePushOptIn(OptInOption.GRANTED);
```

5. Contact data

The main entity in Manago AI is a "**Contact**". A Contact represents a customer/user/website visitor who has provided their email address (the email address is required to create a "Contact Card").

Contacts can be described using a number of properties ("Contact data") useful for various marketing activities, including personalization, segmentation, and targeting. This section describes a number of methods that can be used to transfer Contact data from your mobile app to Manago AI.

NOTE: *When transferring Contact data, the only mandatory field is* ***the email address.*** *If the email address is not transferred, Manago AI can only store the system permission and the marketing consent described in Section 4. above, and the app user is considered anonymous.*

The following Contact properties can be transferred via the Manago AI SDK:

- **Contact data:** name, email address, phone number, standard details (see Sections 5.A and 5.B below)
- **Marketing consents:**
  - Mobile Push and In-App consent (see Section 4 above)
  - Email Marketing consent (see Sections 5.A and 5.C below)
  - Mobile Marketing consent (see Sections 5.A and 5.C below)
- **Custom consents** (see Sections 5.A and 5.D below)
- **Contact tags** (see Sections 5.A and 5.E below)

The respective data fields available in the Manago AI SDK are described in the table in Section 5.F.

The SDK enum class `OptInOption` represents the possible **statuses** for both marketing consents and custom consents:

- `GRANTED`—Contact has given the consent.
- `DENIED`—Contact has not given or has withdrawn the consent.
- `NO_ANSWER`—Contact has neither given nor rejected the consent. The current status will remain unchanged. If there is no status yet, the status will be set to `DENIED`.

5.A. Updating all Contact properties at once

Use the `Salesmanago.updateContactProperties` method to update multiple types of Contact properties at once.

All properties are optional. If you do not want to transfer any of them, specify only the desired ones using named parameters. However, the first time you want to transfer Contact data, you need to include the **email address** (required to create a Contact Card in Manago AI).

TypeScript

```
sdk.updateContactProperties({
  contactData: {
    name: "John Doe",
    email: "john.doe@example.com",
    newEmail: "john.doe.new@example.com",
    phone: "+48123456789",
    userId: "user-123",
    company: "ACME Inc.",
    state: ContactState.CUSTOMER,
    birthday: "19870605",
    streetAddress: "Main Street 1",
    city: "San José",
    zipCode: "90210",
    province: "California",
    country: "US",
    standardDetails: {
      "t_shirt_size": "XL"
    },
    numberDetails: {
      "shoe_size": 10
    },
    dateDetails: {
      "member_since": 1_459_000_000
    },
    doubleOptIn: {
      "emailId": "a2b2-...-c3d4",
      "language": "EN"
    }
  },
  marketingConsents: {
    email: OptInOption.GRANTED,
    mobile: OptInOption.DENIED,
    monitoring: OptInOption.NO_ANSWER
  },
  additionalConsents: [
    { name: "custom consent", status: OptInOption.DENIED }
  ],
  tagsToAdd: ["T_SHIRTS"],
  tagsToRemove: ["FIRST_PURCHASE"],
  setAsNull: ["phone", "company"]
});
```

5.B. Updating basic Contact data

You can use a dedicated method to add or edit only basic Contact properties: name, address, phone number, and standard details.

Additionally, this method passes the email address, meaning it can be used to **create new Contacts**.

TypeScript

```
sdk.updateContactData({
  name: "John Doe",
  email: "john.doe@example.com",
  newEmail: "john.doe.new@example.com",
  phone: "+48123456789",
  userId: "user-123",
  company: "ACME Inc.",
  state: ContactState.CUSTOMER,
  birthday: "19870605",
  streetAddress: "Main Street 1",
  city: "San José",
  zipCode: "90210",
  province: "California",
  country: "US",
  standardDetails: {
    "t_shirt_size": "XL"
  },
  numberDetails: {
    "shoe_size": 10
  },
  dateDetails: {
    "member_since": 1_459_000_000
  },
  doubleOptIn: {
    "emailId": "a2b2-...-c3d4",
    "language": "EN"
  }
});
```

5.C. Updating Email Marketing and Mobile Marketing consent status

The **Email Marketing consent** is required to send marketing emails (newsletter) to your Contacts.

The **Mobile Marketing consent** is required to send marketing text messages (SMS, WhatsApp, Viber) to your Contacts.

Both these consent types can be requested via your mobile app and transferred to Manago AI.

NOTE: *This method can only be used after the user's email address has been transferred to Manago AI using the* `Salesmanago.updateContactProperties` *or* `Salesmanago.updateContactData` *method.*

```
TypeScript
sdk.updateContactMarketingConsents({
  email: OptInOption.DENIED,
  mobile: OptInOption.NO_ANSWER,
  monitoring: OptInOption.GRANTED
});
```

5.D. Updating custom consents

Custom consents are consents other than marketing consents (for example, consent to the processing for personal data for the purposes of an agreement). They are defined individually by each Manago AI Client.

Custom consents are created and managed in **`Menu → Audiences → Contacts → Custom consents.`**

NOTE: *This method can only be used after the user's email address has been transferred to Manago AI using the* `Salesmanago.updateContactProperties` *or* `Salesmanago.updateContactData` *method.*

```
TypeScript
sdk.updateContactAdditionalConsents([
  { name: "some custom consent", status: OptInOption.GRANTED }
]);
```

5.E. Adding and removing Contact tags

Tags are labels assigned to Contacts to enable their segmentation and precise targeting.

Tags can only consist of letters, digits, underscores (_), and dashes (-). Spaces will be converted to underscores. The minimum length is 3 characters, and the maximum length is 255 characters.

NOTE: *This method can only be used after the user's email address has been transferred to Manago AI using the* `Salesmanago.updateContactProperties` *or* `Salesmanago.updateContactData` *method.*

```
TypeScript
sdk.addTags(["tag_to_add"]);
sdk.removeTags(["tag_to_remove"]);
```

5.F. Setting values as null

Setting values to null is done explicitly:

```
TypeScript
sdk.setAsNull(['phone']);
```

5.G. Data field specification

The data fields available in the Manago AI SDK are described in the table below.

The same fields are used in all five methods for transferring Contact properties (Sections 4.A–4.E above).

Fields available in Manago AI SDK

Object	Field	Limits	Description
`contactData`	`name`	255	Contact's name, for example, first and last name.
	`email`	RFC822	Required to create a Contact Card and store Contact data in Manago AI. App users who have not provided an email address are referred to as "anonymous" and the only data that can be stored for them is system permission status and marketing consent status.
	`newEmail`	RFC882	New email address of the Contact. The specified email address cannot be already assigned to another Contact. In Java implementation, leave this field as null to keep the existing email address of the Contact.
	`phone`	Plus sign followed by 11 digits	Contact's phone number. It should start with + followed by the country code.
	`userId`	255	Optional field for user identification on supported Manago AI accounts
	`company`	255	Company the Contact is associated with. In Manago AI, you can list Contacts associated with a given company.
	`state`	enum	Contact's state (CUSTOMER, PROSPECT, PARTNER, OTHER, UNKNOWN)
	`birthday`	YYYYMMDD or MMDD	Contact's birthday (used for birthday emails).
	`streetAddress`	255	Street and apartment number. Additional data is allowed.

	`city`	255	City. Special characters are allowed.
	`zipCode`	255	Zip code (postcode). Special characters are allowed.
	`province`	255	Administrative division within a country. Special characters are allowed.
	`country`	255	Country. Special characters are allowed.
	`standardDetails` (object)	16×255	Object containing key-value pairs. Standard details can be used to store additional information about Contacts.
	`numberDetails` (object)	16×255	Object containing key-value pairs. Number details can be used to store additional information about Contacts.
	`dateDetails` (object)	16×255	Object containing key-value pairs. Date details can be used to store additional information about Contacts or trigger automations at a specific time.
	`doubleOptIn` (object)	`emailId:` UUID `language:` ISO 639-1	Either the ID of the email to be sent or the Contact's language to be used in the double opt-in email
`marketingConsents`	`emailOptIn`	SDK enum class `OptInOption`	Email Marketing consent status: • `GRANTED`—Contact has given the consent. • `DENIED`—Contact has not given or has withdrawn the consent. • `NO_ANSWER`—Contact has neither given nor rejected the consent. The current status will remain unchanged. If there is no status yet, the status will be set to `DENIED`.
	`mobileOptIn`	SDK enum class `OptInOption`	Mobile Marketing consent status: • `GRANTED`—Contact has given the consent. • `DENIED`—Contact has not given or has withdrawn the consent. • `NO_ANSWER`—Contact has neither given nor rejected the consent. The current status will remain unchanged. If there is no status yet, the status will be set to `DENIED`.
`additionalConsents`	array of objects	255 SDK enum class	Array of objects containing the name of a custom consent and its status:

		`OptInOption`	• `GRANTED`—Contact has given the consent. • `DENIED`—Contact has not given or has withdrawn the consent. • `NO_ANSWER`—Contact has neither given nor rejected the consent. The current status will remain unchanged. If there is no status yet, the status will be set to `DENIED`. Custom consents can be created in **Menu → Audiences → Contacts → Custom consents**.
`tagsToAdd`	array	3-255 per tag a-zA-Z0-9_ and - Max. 16 tags per request	Array of tags to be assigned to the Contact. Use ASCII characters only.
`tagsToRemove`	array	3-255 per tag a-zA-Z0-9_ and - Max. 16 tags per request	Array of tags to be removed from the Contact. Use ASCII characters only. **NOTE:** Tags present in both the `addTags` and `removeTags` arrays will be removed only. In that event, Automation Processes based on assigning a tag or increasing a tag scoring will not be triggered.

6. Adding events

The Manago AI SDK enables tracking user activity by transferring predefined events to Manago AI. The events are recorded for both Contacts and anonymous app users.

6.A. App-level events

`EventType` is an SDK enum class representing the possible event types. At present, the following event types are available:

- `LOGIN`—Occurs when the user logs in to your mobile app.

```
TypeScript
sdk.addEvent(EventType.LOGIN);
```

6.B. External Events (transactions)

Adding External Events

Tracking external events with transaction details. External events can be recorded only for contacts with an e-mail. The method `addExternalEvent` returns a Promise resolving to an eventId UUID string which can be used to update or delete the event later. It returns an empty string if the SDK is not properly initialized.

```
TypeScript
const eventId = await sdk.addExternalEvent({
  eventTime: "1772700397",
  eventType: "purchase",
  products: ["P012345", "P123456", "P234567"],
  value: 99.99,
  location: "examplecom",
  externalId: "ID-123456",
  detail1: "Payment method: Credit Card",
  detail2: "Shipping: Express",
  description: "Summer sale order"
});
```

Updating External Events

You can update an existing External Event by providing the eventId returned from `addExternalEvent`. `updateExternalEvent` returns a Promise and rejects when required fields are missing or `eventType` is invalid.

TypeScript

```
await sdk.updateExternalEvent({
  eventId: eventId,
  eventTime: "1772700397",
  eventType: "purchase",
  value: 149.99,
  description: "Updated order value"
});
```

Deleting External Events

You can delete an External Event using `eventId` or `externalId`. At least one must be specified. If both are provided, `eventId` takes priority. The method `deleteExternalEvent` returns a Promise and rejects when both identifiers are missing.

TypeScript

```
await sdk.deleteExternalEvent({ eventId: eventId });
// or by externalId
await sdk.deleteExternalEvent({ externalId: "order-456" });
```

We are here to support you

If you have any questions or doubts concerning the configuration of the Mobile Push channel, or if you would like to have your setup verified by our Support specialist, please contact us at: support@manago.ai

7. Expo Integration

Important notes:

- Expo Go is not supported because the SDK contains native iOS and Android code.
- The app must be built as an Expo development build, EAS build, or native build.
- The Config Plugin prepares native projects during prebuild.
- Runtime application code is still required for SDK initialization and notification permission handling.

7.A. Installation

To install the required packages, run the following command in your project directory:

```shell
yarn add @salesmanago/mobile-push-sdk-react-native npx expo install expo-notifications
```

7.B. Expo Config Plugin configuration

To add the plugin to your project, include the following configuration in your `app.json` or `app.config.js` file:

```json
{
  "expo": {
    "plugins": [
      [
        "@salesmanago/mobile-push-sdk-react-native",
        {
          "apsEnvironment": "development",
          "googleServicesFile": "./google-services.json"
        }
      ]
    ]
  }
}
```

For App Store and TestFlight builds, apsEnvironment should be set to production:

`{ "apsEnvironment": "production" }`

7.C. Plugin options

Option	Type	Default	Description
`apsEnvironment`	"development" or "production"	"development"	APNs environment for iOS push notifications. Set to "production" for App Store and TestFlight builds.
`googleServicesFile`	string		Path to google-services.json, relative to the Expo project root. Required for Android push notifications.
`mavenSecret`	string	ARTIFACT_REGISTRY_MAVEN_SECRET env var	Optional base64 JSON key for the SALESmanago snapshot Maven repository. Real secrets should be provided through environment variables or EAS secrets, not committed in app.json.
`enableRichPushNotifications`	boolean	false	Automatically adds an iOS Notification Service Extension target for displaying images in push notifications.
`notificationServiceExtensionName`	string	SalesmanagoNotificationServiceExtension	Custom name for the Notification Service Extension target. Only relevant when enableRichPushNotifications is true.

7.D. What the plugin configures automatically

iOS

The plugin automatically:

- Adds push notification entitlement.
- Enables remote-notification background mode.
- Injects SALESmanago SDK hooks into AppDelegate.
- Adds launch handler.
- Adds device token forwarding.
- Adds remote notification handler.
- Routes only SALESmanago payloads to the SDK and leaves non-SALESmanago payloads in the app flow.
- When enableRichPushNotifications is enabled, creates a Notification Service Extension target with NotificationService.swift, links SalesmanagoSDK.xcframework, and configures Xcode build settings.

Android

The plugin automatically:

- Adds SALESmanago Maven repositories.
- Configures the Google Services Gradle plugin.
- Copies google-services.json into android/app during prebuild.

7.E. Expo prebuild / native build flow

To prepare your native projects and run the application, use the Expo prebuild and run commands:

```Shell
npx expo prebuild
```

or build directly:

```Shell
npx expo run:ios
npx expo run:android
```

For EAS Build, the plugin configuration should be committed and required secrets should be provided through EAS environment variables or secrets. EAS runs the config plugin during the prebuild/native build step.

7.F. Runtime SDK setup

While the Config Plugin automates native project setup, the application must still initialize the SDK and request notification permissions at runtime. Follow these implementation steps:

- Create and initialize a Salesmanago instance with init() or initWithResult().
- Request notification permissions with expo-notifications.
- Call updatePushNotificationSystemPermissions(granted) whenever permission state changes and when the app returns to foreground.

SDK initialization

```TypeScript
import { Salesmanago } from '@salesmanago/mobile-push-sdk-react-native';
const sdk = new Salesmanago();
await sdk.initWithResult('<API key>');
```

Alternative fire-and-forget initialization:

```TypeScript
sdk.init('<API key>');
```

Push notification permissions in Expo

Implement the following logic to handle push notification permissions within your Expo application using `expo-notifications`:

TypeScript

```typescript
import { useCallback, useEffect, useRef, useState } from 'react';
import * as Notifications from 'expo-notifications';
import { AppState, type AppStateStatus } from 'react-native';
import { Salesmanago } from '@salesmanago/mobile-push-sdk-react-native';

const sdk = new Salesmanago();

export const usePushNotifications = () => {
  const [granted, setGranted] = useState<boolean | null>(null);
  const isRequesting = useRef(false);

  const checkPermission = useCallback(async () => {
    if (isRequesting.current) {
      return;
    }
    isRequesting.current = true;
    try {
      const { status } = await Notifications.getPermissionsAsync();
      if (status === 'granted') {
        setGranted(true);
        return;
      }
      const { status: newStatus } = await Notifications.requestPermissionsAsync();
      setGranted(newStatus === 'granted');
    } finally {
      isRequesting.current = false;
    }
  }, []);

  useEffect(() => {
    const handleAppStateChange = (nextAppState: AppStateStatus): void => {
      if (nextAppState === 'active') {
        checkPermission();
      }
    };
    const subscription = AppState.addEventListener('change', handleAppStateChange);
    return () => subscription.remove();
  }, [checkPermission]);

  useEffect(() => {
    if (granted !== null) {
      sdk.updatePushNotificationSystemPermissions(granted);
    }
  }, [granted]);
```

```
  useEffect(() => {
    checkPermission();
  }, [checkPermission]);
};
```

Then use the hook in the root component:

TypeScript

```
import { usePushNotifications } from './usePushNotifications';

export default function App() {

  usePushNotifications();

  return (

    // your app

  );

}
```

7.G. Rich Push Notifications on iOS in Expo

To enable Rich Push Notifications, such as images on iOS, configure the plugin with the `enableRichPushNotifications` option:

JSON

```
{
  "expo": {
    "plugins": [
      [
        "@salesmanago/mobile-push-sdk-react-native",
        {
          "apsEnvironment": "development",
          "googleServicesFile": "./google-services.json",
          "enableRichPushNotifications": true
        }
      ]
    ]
  }
}
```

Then regenerate native projects:

```shell
npx expo prebuild --clean
```

This automatically:

- Creates the Notification Service Extension target in the Xcode project.
- Generates NotificationService.swift with the required image download logic.
- Links SalesmanagoSDK.xcframework to the extension target.
- Configures build settings.
- Embeds the extension in the main app.

Optional custom extension name:

```
{ "enableRichPushNotifications": true, "notificationServiceExtensionName": "MyCustomExtensionName" }
```

The default extension target name is SalesmanagoNotificationServiceExtension. The extension Bundle Identifier is automatically set as a child of the main app bundle identifier.

Required push payload

For the Notification Service Extension to be triggered, the push payload must include mutable-content: 1.

```json
{
  "aps": {
    "alert": {
      "title": "Title",
      "body": "Body"
    },
    "mutable-content": 1
  }
}
```

7.H. Deep links in Expo

Expo handles deep linking through the `scheme` property in your configuration file:

```
{ "expo": { "scheme": "salesmanago" } }
```

Replace salesmanago with the URL scheme configured in the SALESmanago dashboard. With this configuration, a push notification containing a deep link such as salesmanago://path/to/screen can be routed by Expo Router or React Navigation linking config. No additional native code changes are needed for Expo deep links.

We are here to support you

If you have any questions or doubts concerning the configuration of the Mobile Push channel, or if you would like to have your setup verified by our Support specialist, please contact us at: support@manago.ai