Manago AI
support@manago.ai
+48 533 600 808

Integrate Manago AI with your Mobile App

FLUTTER

Contents

1. Prerequisites

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

- **Minimum supported OS versions:**
  - Android: min. 5.0 Lollipop (API level 21)
  - iOS: min. 15.0 (iPhone versions 6s and above)
- **Minimum supported environment versions:**
  - Flutter: min. 3.3.0
  - Dart: min. 3.6.0
- **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*

`Salesmanago.instance.updateContactProperties`

or

`Salesmanago.instance.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 Manago AI SDK in your mobile app, add the dedicated dependency to the `pubspec.yaml` file in your project, as shown below:

```
None
dependencies:
    salesmanago_mobile_push: 1.X.X
```

You can check the latest SDK version in the Integration Center

Alternatively, you can add the dependency using the console:

```
Shell
flutter pub add salesmanago_mobile_push
```

After adding the Flutter dependency, make sure your project-level `build.gradle` file includes the Manago AI Maven repository.

Open `<project-root>/android/build.gradle` and check for the following block:

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

After completing the configuration in `pubspec.yaml` and `build.gradle`, run the following command to install the dependency:

```
Shell
flutter pub get
```

All methods available in the Manago AI SDK are accessible via the `Salesmanago` singleton class, which is included in the `sales_manago.dart` file:

```dart
import 'package:salesmanago_mobile_push/sales_manago.dart';
```

2.B. Initialization

To initialize the Manago AI SDK, call the `init()` method, which passes the 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**).

```dart
Salesmanago.instance.init(<API key>);
```

2.C. Initialization listener and SDK readiness

You can also check the SDK initialization status and readiness using promise.

`inSuccess` 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.

```dart
final initResult = await Salesmanago.instance.initWithResult(<API key>);
if (initResult.isSuccess) {
  // SDK initialized successfully
} else {
  // initResult.errorMessage contains details when available
}
```

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:

```
Dart
Salesmanago.instance.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

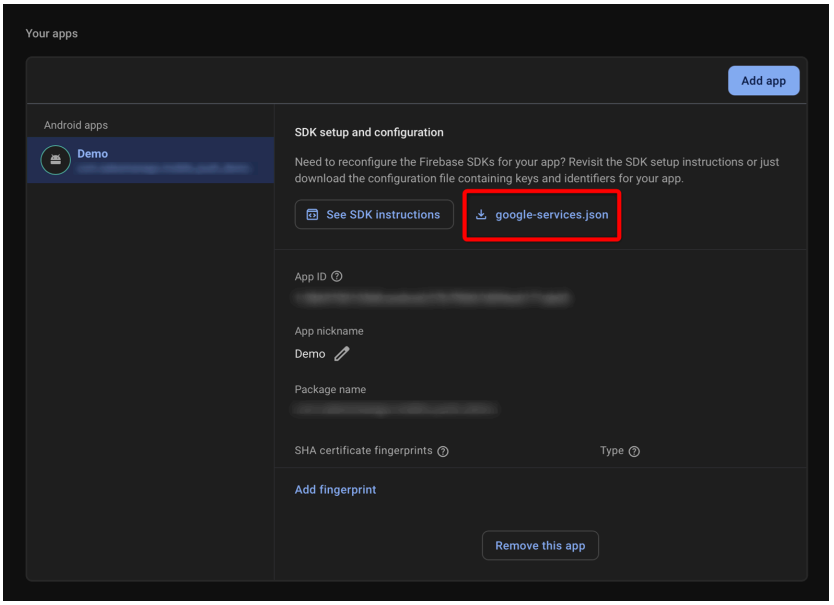
Push notifications are sent via Firebase Cloud Messaging on Android and the Apple Push Notification service (APNs) on iOS. To ensure that your notifications function as intended, perform the configuration steps described below.

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 (`/android/app`).

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 (`/android/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:

```
JavaScript
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:

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

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

    ...
}
```

3.B. 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.

```
Dart
Salesmanago.instance.configure(inAppQueueInterval: 10);
```

3.C. Android: Handling In-App notifications

To make sure In-App notifications are displayed correctly in your Flutter 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.

Notification small icon
To use a dedicated notification icon for Android notifications, add this meta-data entry inside your app application tag in `AndroidManifest.xml`:

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

If the meta-data is not provided, the SDK falls back to the application icon. For best visibility on Android status bar, use a monochrome notification icon (white glyph on transparent background).

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.

Rendering style rules:

If push payload contains a valid imageUrl, notification is shown using BigPictureStyle.

Otherwise, notification is shown using BigTextStyle with the push content.

3.D. Handling multiple push providers

Android

Use this only when you need custom multi-provider split routing. To use a dedicated notification icon for Android notifications, add this meta-data entry inside your app application tag.

Register your own FirebaseMessagingService for multi-provider apps. Route Manago AI payloads using Salesmanago.isSdkNotification(data) and forward only these payloads to SDK handling. To enable this option, replace the SDK service in your app AndroidManifest.xml.

AndroidManifest.xml - custom service

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

Custom FirebaseMessagingService split:

```kotlin
class CustomAppMessagingService : FirebaseMessagingService() {
    private val smService = MessagingService(this)
    override fun onMessageReceived(message: RemoteMessage) {
        if (Salesmanago.isSdkNotification(message.data)) {
            smService.onMessageReceived(message)
        } else {
            // Handle non-Manago AI provider payload.
        }
    }
    override fun onNewToken(token: String) {
        smService.onNewToken(token)
    }
}
```

iOS

When your app uses multiple push providers, route only Manago AI payloads to the SDK. No separate mode is required on iOS. If your app extends FlutterAppDelegate, two valid split patterns are common:

- non-Manago AI → your own flow or super, Manago AI → `Salesmanago.didReceiveRemoteNotification(...)`

- Manago AI → super, non-Manago AI → your own flow (sample app pattern)

In both patterns, keep the same payload split with `Salesmanago.isSdkNotification(...)` and call completionHandler exactly once per notification. If you want to receive non-Manago AI `UNUserNotificationCenter` callbacks (`willPresent` / `didReceive`), register a downstream notification delegate in your `AppDelegate`.

Register downstream delegate:

```
Dart
Salesmanago.setNotificationDelegate(self)
```

If your AppDelegate extends `FlutterAppDelegate`, use this pattern and forward non-Manago AI payloads to super:

```
Dart
public override func application(
    _ application: UIApplication,
    didReceiveRemoteNotification userInfo: [AnyHashable: Any],
    fetchCompletionHandler completionHandler: @escaping (UIBackgroundFetchResult) -> Void
) {
    guard Salesmanago.isSdkNotification(userInfo: userInfo) else {
        // Handle non-Manago AI payload in your own push flow.
        super.application( application, didReceiveRemoteNotification: userInfo,
fetchCompletionHandler: completionHandler )
        return
    }
    Salesmanago.didReceiveRemoteNotification(userInfo: userInfo)
    completionHandler(.noData)
}
```

If you use your own non-Manago AI push flow without forwarding to super, keep the same split but ensure the non-Manago AI path calls completionHandler exactly once in your own code:

```
Dart
public override func application(
    _ application: UIApplication,
    didReceiveRemoteNotification userInfo: [AnyHashable: Any],
    fetchCompletionHandler completionHandler: @escaping (UIBackgroundFetchResult) -> Void
) {
    guard Salesmanago.isSdkNotification(userInfo: userInfo) else {
        // Handle non-Manago AI payload and call completionHandler once in this flow.
        handleNonSalesmanagoPayload(userInfo, completionHandler: completionHandler)
        return
    }
    Salesmanago.didReceiveRemoteNotification(userInfo: userInfo)
    completionHandler(.noData)
}
```

Do not call `completionHandler` on both paths for the same notification.

3.E. Rich Push Notifications (iOS)

To display images in Push notifications on iOS, you need a Notification Service Extension in your Xcode project.

1. Add a Notification Service Extension target

1. In Xcode, go to File > New > Target...
2. Select Notification Service Extension and click Next
3. Enter a product name, for example RichPushNotificationsExtension, and click Finish
4. When prompted to activate the scheme, click Cancel

The extension's Bundle Identifier must be a child of your main app's bundle identifier — for example, if your app is `com.example.myapp`, the extension should be `com.example.myapp.RichPushNotificationsExtension`.

2. Add SalesmanagoSDK to the extension target

1. Select your project in the Project Navigator
2. Select the Notification Service Extension target
3. Go to the General tab
4. Under Frameworks and Libraries, click +
5. Add SalesmanagoSDK.xcframework
6. Make sure the framework is set to Embed & Sign

Also add the extension target to your Podfile so CocoaPods links the SDK into it:

```
None
target 'YourNotificationServiceExtensionTarget' do
  inherit! :search_paths
end
```

3. Implement NotificationService

Replace the entire contents of the generated `NotificationService.swift` file with the implementation below:

```
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.

Apps using UIWindowSceneDelegate

The included example uses AppDelegate, which the plugin handles automatically. If your Runner declares UIApplicationSceneManifest and uses a UIWindowSceneDelegate, forward the scene connection options to the native SDK to preserve cold-start push-click handling:

Note: Ensure the deep link scheme is declared in your Info.plist file as described below.

```swift
import Flutter
import SalesmanagoSDK
import UIKit

final class SceneDelegate: FlutterSceneDelegate {
    override func scene(
        _ scene: UIScene,
        willConnectTo session: UISceneSession,
        options connectionOptions: UIScene.ConnectionOptions
    ) {
        Salesmanago.scene(willConnectWithOptions: connectionOptions)
        super.scene(scene, willConnectTo: session, options: connectionOptions)
    }
}
```

If you use a custom scene delegate instead of FlutterSceneDelegate, keep its existing forwarding of scene lifecycle and URL callbacks to Flutter, then add the Salesmanago.scene(...) call above. Flutter owns navigation for incoming URLs; this call only forwards a notification response to Salesmanago. The SDK keeps the connection options until initialization completes.

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

```xml
<key>FlutterDeepLinkingEnabled</key>
<true/>
<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>
```

You can also specify it in Xcode:

In your application target, open Info tab. In URL Types section click + and enter required data:

- Identifier: your app bundle identifier
- URL scheme: your unique scheme, in this example salesmanago

Now your Flutter application will receive the deeplink main in the navigation configuration:

```dart
MaterialApp(
    onGenerateRoute: (settings) {
        settings.name // this variable should contain 'main'
    },
);
```

Note: The iOS part of the Flutter framework treats deep links like URLs - the navigation works on paths and begins after scheme and host. For instance, the deep link salesmanago://salesmanago.com/main will be truncated to just /main in the Flutter navigation. When creating your deep links, be sure to include the host before path to navigate.
On Android, the app receives the full URL in the native intent.

Flutter navigation policy

The SDK opens the deep link intent, while Flutter owns the navigation stack. Android launchMode settings only control whether Android reuses MainActivity; they do not prevent Flutter from pushing another route for a deep link.

To guarantee that Home remains below a deep link after a cold start, force initialRoute: '/' and dispatch the platform route yourself after the first frame. This avoids relying on the platform-specific representation of Navigator.defaultRouteName:

```dart
final navigatorKey = GlobalKey<NavigatorState>();
@override
void initState() {
    super.initState();
    WidgetsBinding.instance.addPostFrameCallback((_) {
        final routeName = WidgetsBinding.instance.platformDispatcher.defaultRouteName;
        if (isSupportedDeepLink(routeName)) {
            navigatorKey.currentState?.pushNamed(routeName);
        }
    });
}
// In MaterialApp: navigatorKey: navigatorKey, initialRoute: '/',
```

Android cold-start race: when a push notification is tapped while the app is killed, the native click handler launches the app and then delivers the deep link as a separate intent right after. Both resolve to the same MainActivity, so the second intent can arrive before Flutter's engine has attached its own route-delivery channel, and defaultRouteName above may never see it. Call Salesmanago.instance.getInitialDeepLink() once after your first frame as a fallback - it reads the intent independently of that channel, so it still finds the link when defaultRouteName misses it:

```dart
WidgetsBinding.instance.addPostFrameCallback((_) async {
    final routeName = WidgetsBinding.instance.platformDispatcher.defaultRouteName;
    if (isSupportedDeepLink(routeName)) {
        navigatorKey.currentState?.pushNamed(routeName);
        return;
    }
    final pendingLink = await Salesmanago.instance.getInitialDeepLink();
    if (pendingLink != null && isSupportedDeepLink(pendingLink)) {
        navigatorKey.currentState?.pushNamed(pendingLink);
    }
});
```

getInitialDeepLink() consumes the link it returns - a second call returns null until a new one arrives. This fallback is Android-specific; on iOS defaultRouteName alone is sufficient.

Decide how your app should handle a deep link to its current screen. For example, an onGenerateRoute implementation can resolve a supported URL to an internal route while ignoring tracking parameters:

```dart
String? resolveDeepLinkTarget(String? routeName) {
    final uri = Uri.tryParse(routeName ?? '');
    switch (uri?.path) {
        case '/account': return '/account';
        case '/orders': return '/orders';
        default: return null;
    }
}
```

The sample replaces only its technical resolver route with the deep link target. It preserves existing routes in the stack, matching the native Android behavior. Applications that want to deduplicate an existing target route must implement that policy themselves:

```dart
class DeepLinkNavigationArgs {
    const DeepLinkNavigationArgs({ this.isResolved = true, this.queryParameters = const {},
});
    final bool isResolved;
    final Map<String, String> queryParameters;
}

class DeepLinkResolverPage extends StatefulWidget {
    const DeepLinkResolverPage({ required this.targetRoute, required this.queryParameters,
super.key, });
    final String targetRoute;
    final Map<String, String> queryParameters;
    @override State<DeepLinkResolverPage> createState() => _DeepLinkResolverPageState();
}

class _DeepLinkResolverPageState extends State<DeepLinkResolverPage> {
    @override
    void initState() {
        super.initState();
        WidgetsBinding.instance.addPostFrameCallback((_) {
            if (mounted) {
                Navigator.of(context).pushReplacementNamed(
                    widget.targetRoute,
                    arguments: DeepLinkNavigationArgs(
                        queryParameters: widget.queryParameters,
                    ),
                );
            }
        });
    }
    @override
    Widget build(BuildContext context) => const SizedBox.shrink();
}
```

Create the resolver with uri.queryParameters from the parsed URL. The target page can then read them from ModalRoute.of(context)!.settings.arguments as DeepLinkNavigationArgs, for example args.queryParameters['id'] or UTM parameters.

Use this resolver only for URLs your app supports, and map the URL to an internal route before creating it. When deciding whether to create the resolver, do not classify a route with resolved arguments as another deep link:

```dart
final args = settings.arguments;
if (args is DeepLinkNavigationArgs && args.isResolved) {
    return false;
}
```

The same route name can be used by platform deeplinks and local Navigator.pushNamed calls. For a local navigation to a route that is also a deep link target, pass const DeepLinkNavigationArgs() to mark it as already resolved and prevent an unintended stack reset:

```dart
Navigator.of(context).pushNamed(
    '/orders',
    arguments: const DeepLinkNavigationArgs(),
);
```

With this policy, a deep link to the currently displayed screen adds another instance of that screen. Back returns to the previous screen instance. Your application can instead choose a custom policy, such as ignoring a deep link to the current route or removing routes above an existing target.

Recommended integration for apps with a single Flutter Activity

For Flutter applications using a single `Activity` on Android, follow these recommendations to ensure deep links are handled correctly without resetting the application state.

Android Configuration

Set `android:launchMode="singleTask"` on your Flutter Activity in the `AndroidManifest.xml`. Then, override the `onNewIntent()` method in your Activity class to update the intent:

```dart
@Override
protected void onNewIntent(Intent intent) {
    super.onNewIntent(intent);
    setIntent(intent);
}
```

Flutter Navigation Strategy

To ensure the Home page remains in the navigation stack when a deep link is opened at startup, force `initialRoute: '/'` in your `MaterialApp` or `CupertinoApp`. Dispatch the platform route manually from `defaultRouteName` inside an `addPostFrameCallback`.

Implement a `DeepLinkResolverPage` pattern that preserves the existing navigation stack to match native Android behavior. This approach should include an explicit opt-in for local `Navigator.pushNamed` calls when navigating to the same route name, preventing unnecessary stack resets.

Handling Cold-Start Race Conditions (Android)
When a push notification is tapped while the app is killed (cold start), Android fires a launch intent followed by an `ACTION_VIEW` intent. On Android, the second intent can occasionally arrive before the Flutter engine is ready, causing the deep link to be dropped. To fix this, use the `getInitialDeepLink()` method, which captures and stores the intent data independently of the engine's state.

```dart
final pendingLink = await Salesmanago.instance.getInitialDeepLink();
```

Call this method once after the first frame as a fallback if `defaultRouteName` does not resolve to the expected link. Do not call this method on iOS (you can use `Platform.isAndroid` as a condition). Note that iOS is unaffected by this race condition, and standard navigation remains sufficient there.

3.F. 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.

```dart
Future<void> await Salesmanago.instance.configure(
  urlPresentation: UrlPresentation.inAppBrowser,
);
```

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

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 `onPushNotificationSystemPermissionsChanged()` 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.

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 `onPushNotificationSystemPermissionsChanged()` method to update the permission status.

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

For iOS devices, 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.instance.updateMobilePushOptIn(OptInOption)` method. An example is provided below, and the **possible statuses are described in Section 5 below**.

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.*

```dart
Salesmanago.instance.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.
- `noAnswer`—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.instance.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).

NOTE: This method passes the email address, meaning it can be used to **create new Contacts**.

Dart

```dart
import 'package:salesmanago_mobile_push/model/additional_consent.dart';
import 'package:salesmanago_mobile_push/model/contact_data.dart';
import 'package:salesmanago_mobile_push/model/marketing_consents.dart';
import 'package:salesmanago_mobile_push/model/opt_in_option.dart';

Salesmanago.instance.updateContactProperties(
  contactData: ContactData(
    name: 'John Doe',
    email: 'john.doe@example.com',
    newEmail: 'john.doe.new.email@example.com',
    phone: '+15554443322',
    userId: 'U12345',
    company: 'ACME Inc.',
    state: ContactState.customer,
    birthday: '19870605',
    streetAddress: 'Main Street 1',
    city: 'San Jos&#233;',
    zipCode: '90210',
    province: 'California',
    country: 'US',
    standardDetails: {
      'eco_program_member': 'yes',
      't_shirt_size': 'XL',
    },
    numberDetails: {
      'shoe_size': 10,
    },
    dateDetails: {
      'member_since': 1459938888,
    },
    doubleOptIn: DoubleOptIn(
      emailId: 'a2b2-...-c3d4',
      language: 'ES',
    ),
  ),
  marketingConsents: MarketingConsents(
    email: OptInOption.granted,
    mobile: OptInOption.denied,
    monitoring: OptInOption.granted,
  ),
  additionalConsents: [
    AdditionalConsent(
      name: 'TERMS_OF_SERVICE',
      status: OptInOption.denied,
    ),
  ],
  tagsToAdd: ['T_SHIRTS'],
  tagsToRemove: ['FIRST_PURCHASE'],
  setAsNull: ['phone', 'province'],
);
```

Updating only specific types of Contact properties

To facilitate the development of your mobile app, Manago AI provides you with additional methods for updating only specific types of Contact properties (Sections 4.B–4.E). This way, you can avoid writing lengthy code blocks with null parameters in Java.

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**.

```dart
Salesmanago.instance.updateContactData(
  name: 'John Doe',
  email: 'john.doe@example.com',
  newEmail: 'john.doe.new.email@example.com',
  phone: '+15554443322',
  userId: 'U12345',
  company: 'ACME Inc.',
  state: ContactState.customer,
  birthday: '19870605',
  streetAddress: 'Main Street 1',
  city: 'San Jos&#233;',
  zipCode: '90210',
  province: 'California',
  country: 'US',
  doubleOptIn: DoubleOptIn(
    emailId: 'a2b2-c3d4',
    language: 'ES',
  ),
  standardDetails: {
    'eco_program_member': 'yes',
    't_shirt_size': 'XL',
  },
  numberDetails: {
    'shoe_size': 10,
  },
  dateDetails: {
    'member_since': 1459938888,
  },
);
```

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.instance.updateContactProperties` *or* `Salesmanago.instance.updateContactData` *method.*

```dart
Salesmanago.instance.updateContactMarketingConsents(
  email: OptInOption.denied,
  mobile: OptInOption.noAnswer,
);
```

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.

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

```dart
Salesmanago.instance.updateContactAdditionalConsents([
  AdditionalConsent(
    name: 'demo 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, and 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.instance.updateContactProperties` *or* `Salesmanago.instance.updateContactData` *method.*

```dart
Salesmanago.instance.addTags(['only_tag_to_add']);
Salesmanago.instance.removeTags(['only_tag_to_remove']);
```

5.F. Setting values as null

Setting values to null is done explicitly:

```
Dart
Salesmanago.instance.setAsNull(List<String> fieldsToSetNull);
```

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 5.A–5.E above).

Fields available in Manago AI SDK

Object [Kotlin] / Parameter number [Java]	Field	Limits	Description
`contactData` / First parameter	`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`	RFC822	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` / Second parameter	`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` / Third parameter	array of objects	255 SDK enum	Array of objects containing the name of a custom consent and its status:

		class `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` / Fourth parameter	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` / Fifth parameter	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.

```kotlin
import 'package:salesmanago_mobile_push/model/event_type.dart';

Salesmanago.instance.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.

```dart
import 'package:salesmanago_mobile_push/model/external_event_type.dart';
final eventId = await Salesmanago.instance.addExternalEvent(
  eventTime: '1772700397',
  eventType: ExternalEventType.purchase,
  products: ['product1', 'product2', 'product3'],
  value: 99.99,
  location: 'shop123',
  externalId: 'order-456',
  detail1: 'Payment method: Credit Card',
  detail2: 'Shipping: Express',
  detail3: 'Discount: 10%',
  detail4: 'Customer segment: Premium',
  detail5: 'Campaign: Summer Sale',
  description: 'Example purchase transaction',
);
```

Updating and deleting External Events

While adding a new external event, the SDK generates an eventId (UUID) and returns it for further managing the event via other methods. This can be used to update or delete the event via updateExternalEvent and deleteExternalEvent methods:

```dart
Salesmanago.instance.updateExternalEvent(
  eventId: 'fe964947-23b6-4950-a8a0-1848c20baf78',
  eventTime: '1772700397',
  eventType: ExternalEventType.cancellation,
  products: ['product1', 'product2', 'product3'],
  value: 1.0,
  detail1: 'Updated detail',
  description: 'Updated description',
);
```

Method deleteExternalEvent uses eventId or externalId to identify the event. At least one of the params has to be specified. If you specify both, the eventId will take priority.

```dart
Salesmanago.instance.deleteExternalEvent(
  eventId: 'fe964947-23b6-4950-a8a0-1848c20baf78',
  externalId: 'fe964947-23b6-4950-a8a0-1848c20baf78',
);
```

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