Integrate Manago AI with your Mobile App

ANDROID (Native)

Contents

1. Prerequisites

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

- **Minimum supported OS Version:** The minimum Android version supported by the Manago AI SDK is 5.0 Lollipop (API level 21).
- **google-services.json:** The google-services.json file (downloaded from your Firebase account) must be placed in the app-level root directory (see Section 3.A below).
- **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—for Android 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 3 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.updateContactProperties` *or* `Salesmanago.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 build.gradle file in your project, as shown below. You should always check the latest version of the SDK available in

```
None
implementation("com.salesmanago:mobilepush:1.X.X")
```

Next, add a custom Maven location in the repositories {} section:

```
None
repositories {
    // ...

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

Note on code snippets

This guide provides code snippets in both Java and Kotlin to support diverse Android development environments. The applicable language is indicated in the top-left corner of each block.

2.B. Initialization

To initialize the Manago AI SDK, call the `init()` method, passing the `applicationContext` object and the Manago AI API key. This method must be called at the app's startup, usually in the `onCreate` method of the `Application` class.

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

Kotlin
```
Salesmanago.init(applicationContext, <Manago AI API key>)
```

Java
```
Salesmanago.INSTANCE.init(applicationContext, <Manago AI API key>);
```

2.C. Initialization listener and SDK readiness

You can also pass InitializationListener to receive initialization completion callbacks.

`onSuccess()` 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.

Kotlin
```
Salesmanago.init(applicationContext, <API key>, object : InitializationListener {
    override fun onSuccess() {
        // SDK is ready
    }
    override fun onError(error: Throwable) {
        // Core initialization failed
    }
})
```

Java
```
Salesmanago.INSTANCE.init(applicationContext, <API key>, new InitializationListener() {
    @Override
    public void onSuccess() {
        // SDK is ready
    }
    @Override
    public void onError(Throwable error) {
        // Core initialization failed
    }
});
```

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:

Kotlin
```
Salesmanago.reInit(<new API key>)
```

Java
```
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

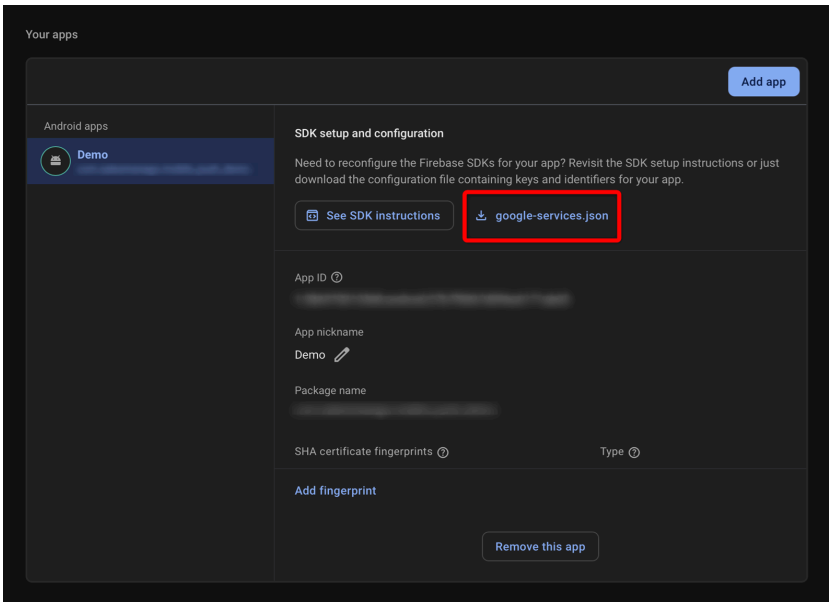
Push notifications are sent using Firebase Cloud Messaging. The Manago AI SDK handles all Firebase management tasks for you.

3.A. 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.

See the instructions in Firebase's help portal >>

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

```
None
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:

```
None
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.

```
Kotlin
Salesmanago.configure(inAppQueueInterval = 10)
```

```
Java
Salesmanago.INSTANCE.configure(10);
```

3.C. 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

Starting from Android version 13, 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, for example, using the `ActivityCompat.requestPermissions` method.

When the system permission is granted or denied, this status must be explicitly transferred to Manago AI using the `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.

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

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

Additionally, every time you call the `init` method, 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).

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.

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.*

Kotlin

```kotlin
object PushConsentManager {

    fun askForMarketingConsent(context: Context) {
        AlertDialog.Builder(context)
            .setTitle("Marketing Consent")
            .setMessage("Would you like to receive marketing notifications?")
            .setPositiveButton("Yes") { _, _ ->
                Salesmanago.updateMobilePushOptIn(OptInOption.GRANTED)
            }
            .setNegativeButton("No") { _, _ ->
                Salesmanago.updateMobilePushOptIn(OptInOption.DENIED)
            }
            .setNeutralButton("Not now") { _, _ ->
                Salesmanago.updateMobilePushOptIn(OptInOption.NO_ANSWER)
            }
            .show()
    }
}
```

Java

```java
public class PushConsentManager {

    public static void askForMarketingConsent(Context context) {
        new AlertDialog.Builder(context)
            .setTitle("Marketing Consent")
            .setMessage("Would you like to receive marketing notifications?")
            .setPositiveButton("Yes", (dialog, which) -> {
                Salesmanago.updateMobilePushOptIn(OptInOption.GRANTED);
            })
            .setNegativeButton("No", (dialog, which) -> {
                Salesmanago.updateMobilePushOptIn(OptInOption.DENIED);
            })
            .setNeutralButton("Not now", (dialog, which) -> {
                Salesmanago.updateMobilePushOptIn(OptInOption.NO_ANSWER);
            })
            .show();
    }
}
```

3.D. Link handling

Deep links

Unlike standard links that 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 your Mobile Push and/or In-App notifications, you need to declare them in your app.

For instance, if you create a notification with the following deep link: `manago-ai://main`, the destination screen must declare `host` and `scheme` in the `AndroidManifest.xml` file, as shown below:

```xml
<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:host="main"
        android:scheme="manago-ai" />
</intent-filter>
```

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, giving a more integrated experience without leaving the app.

```kotlin
import com.salesmanago.library.Salesmanago
import com.salesmanago.library.model.PushUrlPresentation
// Open HTTP(S) URLs in an Android Custom Tab.
Salesmanago.configurePushUrlPresentation(
    PushUrlPresentation.IN_APP_BROWSER
)
// Restore the default external-browser behavior.
Salesmanago.configurePushUrlPresentation(
    PushUrlPresentation.EXTERNAL_BROWSER
)
```

```java
import com.salesmanago.library.Salesmanago;
import com.salesmanago.library.model.PushUrlPresentation;
// Open HTTP(S) URLs in an Android Custom Tab.
Salesmanago.INSTANCE.configurePushUrlPresentation(
    PushUrlPresentation.IN_APP_BROWSER
);
// Restore the default external-browser behavior.
Salesmanago.INSTANCE.configurePushUrlPresentation(
    PushUrlPresentation.EXTERNAL_BROWSER
);
```

3.E. Custom FCM Service

The SDK provides an internal FCM service (called MessagingService) that is public and can be used in a custom notification service. The internal service is registered automatically, but you can create your own service and disable the automatic registration.

When integrating a custom FCM service, use Salesmanago.isSdkNotification(data) in Kotlin or Salesmanago.INSTANCE.isSdkNotification(data) in Java to check whether a message comes from SALESmanago before forwarding it to MessagingService.

```kotlin
import com.google.firebase.messaging.FirebaseMessagingService
import com.google.firebase.messaging.RemoteMessage
import com.salesmanago.library.Salesmanago
import com.salesmanago.library.common.messaging.MessagingService
class CustomAppMessagingService : FirebaseMessagingService() {
    private val smService by lazy { MessagingService(this) }
    override fun onMessageReceived(message: RemoteMessage) {
        // Your custom logic before SDK processing
        // ...
        if (Salesmanago.isSdkNotification(message.data)) {
            smService.onMessageReceived(message)
        }
        // Your custom logic after SDK processing
    }
}
```

Java

```java
import com.google.firebase.messaging.FirebaseMessagingService;
import com.google.firebase.messaging.RemoteMessage;
import com.salesmanago.library.Salesmanago;
import com.salesmanago.library.common.messaging.MessagingService;
public class CustomAppMessagingService extends FirebaseMessagingService {
    private MessagingService smService;
    @Override
    public void onCreate() {
        super.onCreate();
        smService = new MessagingService(this);
    }
    @Override
    public void onMessageReceived(RemoteMessage message) {
        // Your custom logic before SDK processing
        // ...
        if (Salesmanago.INSTANCE.isSdkNotification(message.getData())) {
            smService.onMessageReceived(message);
        }
        // Your custom logic after SDK processing
    }
}
```

Creating a Custom FCM Service

The SDK's internal FCM service is public and can be instantiated by providing a Context. You can then call its methods as needed.

Kotlin

```kotlin
import com.google.firebase.messaging.FirebaseMessagingService
import com.google.firebase.messaging.RemoteMessage
import com.salesmanago.library.common.messaging.MessagingService
class CustomAppMessagingService : FirebaseMessagingService() {
    private val smService = MessagingService(this)
    override fun onMessageReceived(message: RemoteMessage) {
        // Your custom logic before SDK processing
        // ...
        smService.onMessageReceived(message)
        // Your custom logic after SDK processing
    }
    override fun onNewToken(token: String) {
        // Your custom logic before SDK processing
        // ...
        smService.onNewToken(token)
        // Your custom logic after SDK processing
    }
}
```

```java
import com.google.firebase.messaging.FirebaseMessagingService;
import com.google.firebase.messaging.RemoteMessage;
import com.salesmanago.library.common.messaging.MessagingService;
public class CustomAppMessagingService extends FirebaseMessagingService {
    private final MessagingService smService = new MessagingService(this);
    @Override
    public void onMessageReceived(RemoteMessage message) {
        // Your custom logic before SDK processing
        // ...
        smService.onMessageReceived(message);
        // Your custom logic after SDK processing
    }
    @Override
    public void onNewToken(String token) {
        // Your custom logic before SDK processing
        smService.onNewToken(token);
        // Your custom logic after SDK processing
    }
}
```

Disabling Automatic SDK Service Registration

To use your custom service, you must disable the automatic registration of the SDK's internal service in AndroidManifest.xml. Use the `tools:node="replace"` attribute in your service declaration:

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">
    <application>
        <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>
```

The `tools:node="replace"` attribute causes your service to replace the service registered by the SDK, disabling the automatic registration of the internal service.

3.F. Push notification style

Notification content style is selected automatically by the SDK:

- For text-only Push notifications, the SDK uses expanded text (`BigTextStyle`) so longer content can be read after expanding the notification,
- For Push notifications with an image, the SDK uses BigPictureStyle when the image is available.

Small icon

If you want to use a dedicated small icon for Push notifications (instead of the app icon), add the following meta-data to your app-level `AndroidManifest.xml` inside the application tag:

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

Replace `@drawable/your_push_icon` with your own notification small icon resource. The icon should be a monochromatic notification small icon suitable for the Android status bar. If this value is not provided, the SDK first checks com.google.firebase.messaging.default_notification_icon and then falls back to the application icon.

4. 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 3.B. 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 4.A and 4.B below)
- **Marketing consents:**
    - Mobile Push and In-App consent (see Section 3 above)
    - Email Marketing consent (see Section 4.C below)
    - Mobile Marketing consent (see Section 4.C below)
- **Custom consents** (see Section 4.D below)
- **Contact tags** (see Section 4.E below)

The respective data fields available in the Manago AI SDK are described in the table in Section 4.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`.

4.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, use named parameters in Kotlin or set these parameters to null in Java. 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).

Kotlin

```kotlin
Salesmanago.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", // YYYYMMDD or MMDD
        streetAddress = "Main Street 1",
        city = "San José",
        zipCode = "90210",
        province = "California",
        country = "US",
        standardDetails = mapOf(
            "eco_program_member" to "yes",
            "t_shirt_size" to "XL"
        ),
        numberDetails = mapOf(
            "shoe_size" to 10
        ),
        dateDetails = mapOf(
            "member_since" to 1459938888
        ),
        doubleOptIn = DoubleOptIn(
            emailId = "a2b2-...-c3d4",
            language = "es" //Include either the language or emailId
        )
    ),
    marketingConsents = MarketingConsents(
        email = OptInOption.GRANTED,
        mobile = OptInOption.DENIED,
       monitoring = OptInOption.GRANTED,
    ),
    additionalConsents = listOf(
        AdditionalConsent(
            "TERMS_OF_SERVICE",
            OptInOption.DENIED
        )
    ),
    tagsToAdd = listOf("T_SHIRTS"),
    tagsToRemove = listOf("FIRST_PURCHASE"),
    setAsNull = listOf("phone", "province")
)
```

Argument order change
The `newEmail` parameter has been added as the third argument, which shifts subsequent parameters. Ensure your implementation is updated if migrating from an older version of the SDK.

Java

```
Salesmanago.INSTANCE.updateContactProperties(
    new ContactData(
        "John Doe",
        "john.doe@example.com",
        "john.doe.new.email@example.com",
        "+15554443322",
        "U12345", //use null if you don't use User ID
        "ACME Inc.",
        ContactState.CUSTOMER,
        "19870605",
        "Main Street 1",
        "San José",
        "90210",
        "California",
        "US",
        new DoubleOptIn(
            "a2b2-...-c3d4",
            "es" //Include either the language or emailId, setting the other to null
        ),
        new HashMap<String, String>() {
            put("eco_program_member", "yes");
            put("t_shirt_size", "XL");
        },
        new HashMap<String, Integer>() {
            put("shoe_size", 10);
        },
        new HashMap<String, Integer>() {
            put("member_since", 1459938888);
        }
    ),
    new MarketingConsents(
        OptInOption.DENIED, //Email Marketing
        OptInOption.NO_ANSWER, //Mobile Marketing
        OptInOption.GRANTED //Website monitoring. This field may be removed in the future
    ),
    Arrays.asList(
        new AdditionalConsent(
            "TERMS_OF_SERVICE", OptInOption.DENIED
        )
    ),
    Arrays.asList("T_SHIRTS"), //Tags to add
    Arrays.asList("FIRST_PURCHASE"), //Tags to remove
    Arrays.asList("phone", "province") //Set as null
);
```

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.

4.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**.

Kotlin

```
Salesmanago.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", // YYYYMMDD or MMDD
    streetAddress = "Main Street 1",
    city = "San José",
    zipCode = "90210",
    province = "California",
    country = "US",
    standardDetails = mapOf(
        "eco_program_member" to "yes",
        "t_shirt_size" to "XL"
    ),
    numberDetails = mapOf(
        "shoe_size" to 10
    ),
    dateDetails = mapOf(
        "member_since" to 1459938888
    ),
    doubleOptIn = DoubleOptIn(
        emailId = "a2b2-...-c3d4",
        language = "es" //Include either the language or emailId
    )
)
```

Argument order change

The `newEmail` parameter has been added as the third argument, which shifts subsequent parameters. Ensure your implementation is updated if migrating from an older version of the SDK.

```java
Salesmanago.INSTANCE.updateContactData(
    "John Doe",
    "john.doe@example.com",
    "john.doe.new.email@example.com",
    "+15554443322",
    "U12345", //use null if you don't use User ID
    "ACME Inc.",
    ContactState.CUSTOMER,
    "19870605",
    "Main Street 1",
    "San José",
    "90210",
    "California",
    "US",
    new DoubleOptIn(
        "a2b2-...-c3d4",
        "es" //Include either the language or emailId, setting the other to null
    ),
    new HashMap<String, String>() {
        put("eco_program_member", "yes");
        put("t_shirt_size", "XL");
    },
    new HashMap<String, Integer>() {
        put("shoe_size", 10);
    },
    new HashMap<String, Integer>() {
        put("member_since", 1459938888);
    }
);
```

4.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* `updateContactProperties` *or* `updateContactData` *method.*

```kotlin
Salesmanago.updateContactMarketingConsents(
```

```
    email = OptInOption.DENIED,
    mobile = OptInOption.NO_ANSWER,
    monitoring = OptInOption.GRANTED
)
```

Java

```
Salesmanago.INSTANCE.updateContactMarketingConsents(
    OptInOption.DENIED, //Email Marketing
    OptInOption.NO_ANSWER, //Mobile Marketing
    OptInOption.GRANTED //Website monitoring*
);
```

* This field, corresponding to the user's consent to website monitoring (the storage of a Manago AI cookie in their browser), may be removed from the SDK after the beta phase.

4.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* `updateContactProperties` *or* `updateContactData` *method.*

Kotlin

```
Salesmanago.updateContactAdditionalConsents(
    listOf(
        AdditionalConsent(
            "demo custom consent",
            OptInOption.GRANTED
        )
    )
)
```

Java

```
Salesmanago.INSTANCE.updateContactAdditionalConsents(
    Arrays.asList(
        new AdditionalConsent(
            "demo custom consent",
            OptInOption.GRANTED
        )
    )
);
```

4.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 (-). Tags containing other characters will not be accepted by the SDK. 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* `updateContactProperties` *or* `updateContactData` *method.*

Kotlin
```
Salesmanago.addTags(listOf("tag_to_add"))
Salesmanago.removeTags(listOf("tag_to_remove"))
```

Java
```
Salesmanago.INSTANCE.addTags(Arrays.asList("tag_to_add"));
Salesmanago.INSTANCE.removeTags(Arrays.asList("tag_to_remove"));
```

4.F. Setting values as null

Setting values to null is done explicitly:

Kotlin
```
Salesmanago.setAsNull(listOf("phone"))
```

Java
```
Salesmanago.INSTANCE.setAsNull(Arrays.asList("phone"));
```

4.F. 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 [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`	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` / 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 class `OptInOption`	Array of objects containing the name of a custom consent and its 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`.

			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.

5. 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.

5.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
```
Salesmanago.addEvent(EventType.LOGIN)
```

Java
```
Salesmanago.INSTANCE.addEvent(EventType.LOGIN);
```

5.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.

Kotlin
```
val eventId = Salesmanago.addExternalEvent(
    eventTime = "1772700397",
    eventType = ExternalEventType.PURCHASE,
    products = listOf("P012345", "P123456", "P234567"),
    value = 99.99F,
    location = "examplecom",
    externalId = "ID-123456",
    detail1 = "Payment method: Credit Card",
    detail2 = "Shipping: Express",
    detail3 = "Discount: 10%",
    detail4 = "Customer segment: Premium",
    detail5 = "Campaign: Summer Sale",
    description = "Example purchase transaction"
)
```

Java
```
String eventId = Salesmanago.INSTANCE.addExternalEvent(
    "1772700397", //Unix timestapm
    ExternalEventType.PURCHASE, //Type
    Arrays.asList("P012345", "P123456", "P234567"), //Product IDs
    99.99F, //Value
    "examplecom", //Location field
    "ID-123456", //External ID (e.g. order ID)
    "Payment method: Credit Card", //Detail1
    "Shipping: Express", //Detail2
    "Discount: 10%", //Detail3
    "Customer segment: Premium", //Detail4
    "Campaign: Summer Sale", //Detail5
    // remaining details 6-20/ or null values
    "Example purchase transaction" //Description
);
```

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:

Kotlin
```
Salesmanago.updateExternalEvent(
    eventId = "fe964947-23b6-4950-a8a0-1848c20baf78",
    eventTime = "1772700397",
    eventType = ExternalEventType.CANCELLATION,
    products = listOf("P012345", "P123456", "P234567"),
    value = 1F,
    detail1 = "Updated detail",
    description = "Updated description"
)
```

Java
```
Salesmanago.INSTANCE.updateExternalEvent(
    "fe964947-23b6-4950-a8a0-1848c20baf78",
    "1772700397",
    ExternalEventType.CANCELLATION,
    Arrays.asList("P012345", "P123456", "P234567"),
    1F,
    "Updated detail",
    // rest of the details 2-20
    "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.

Kotlin

```
Salesmanago.deleteExternalEvent(
    eventId = "fe964947-23b6-4950-a8a0-1848c20baf78",
    externalId = "ID-123456"
)
```

Java

```
Salesmanago.INSTANCE.deleteExternalEvent(
    "fe964947-23b6-4950-a8a0-1848c20baf78",
    "ID-123456"
);
```

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