SDK

NoPass Passwordless Registration and Authentication

Prepared By:

Identité™, Inc.
3035 Turtle Brooke
Clearwater, Florida 33761 USA
www.Identite.us

Prepared For:

Version: **0.3**
Dated: **09 September 2020**

REVISION HISTORY			
REVISION	**REVISION**	**REVISION**	**REVISION**
0.1	Tatsiana Kachan	Initial template	27 July 2020
0.2	Vitaly Moroz	Author of the document	01 September 2020
0.3	Elena Dubinenko	Review, copy-editing and proofreading.	09 September 2020

SIGNATURE

This **NoPass SDK** documents the request for development of the NoPass SDK Project to be performed by Identité™, Inc., for **Identité™, Inc.** (“Customer”).

APPROVED BY:

Identité™, Inc.	**Date**

PREPARED BY:

Identité™, Inc.	**Date**

CONTENTS

OVERVIEW

NoPass mobile application SDK serves to connect your existing application to the NoPass passwordless authentication system. There are few main components:

- NoPassRegistrationManager
- NoPassUserManager
- NoPassUtils

INITIAL SETUP

Procedure

1. Add the library file to your libs folder.
2. Add the dependencies to your module level gradle build file:

```
implementation "com.android.support:multidex:1.0.2"
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.3.0"
implementation 'androidx.appcompat:appcompat:1.1.0'
implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
implementation(name: 'nopass-core-release', ext: 'aar') {
   exclude group: 'org.jetbrains.kotlin', module: 'kotlin-reflect'
}
implementation 'io.reactivex.rxjava2:rxjava:2.2.0'
implementation "io.reactivex.rxjava2:rxkotlin:2.2.0"
implementation "io.reactivex.rxjava2:rxandroid:2.0.0"
implementation "com.squareup.retrofit2:retrofit:2.3.0"
implementation "com.squareup.okhttp3:logging-interceptor:3.9.0"
implementation "com.squareup.retrofit2:converter-moshi:2.3.0"
implementation "com.squareup.retrofit2:adapter-rxjava2:2.3.0"
implementation("com.squareup.moshi:moshi-kotlin:1.5.0")
implementation "com.serjltt.moshi:moshi-lazy-adapters:2.1"
implementation "com.android.support:multidex:1.0.2"
implementation 'com.google.code.gson:gson:2.8.5'
implementation "androidx.room:room-runtime:2.1.0-alpha06"
implementation "androidx.room:room-rxjava2:2.1.0-alpha06"
implementation "com.scottyab:rootbeer-lib:0.0.7"
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.3.0"
implementation 'com.google.http-client:google-http-client-gson:1.26.0'
implementation 'com.google.api-client:google-api-client-android:1.30.8'
implementation 'com.google.apis:google-api-services-drive:v3-rev194-
1.25.0'
implementation 'com.google.firebase:firebase-auth:19.2.0'
kapt "androidx.room:room-compiler:2.1.0-alpha06"
implementation 'com.google.firebase:firebase-messaging:20.2.3'
implementation 'com.google.firebase:firebase-analytics:17.4.4'
```

3. To make core functionality work, hook the NoPass service to the firebase callbacks and provide the application instance.

Note: NoPass uses the Firebase cloud messaging system in its registration and authorization flows. So, in order to complete them successfully, you will have to set up the so-called *interceptors* inside your Firebase `onMessageReceived()` callback.

Simply provide the data for further processing. Any non-NoPass data will be ignored:

```
override fun onMessageReceived(remoteMessage: RemoteMessage) {
   interceptFirebaseMessage(remoteMessage.data)
}
```

Make sure the firebase token is provided correctly and will be available during registration and authorization flows:

```
interceptFirebaseToken(token)
```

4. And, finally, give us your application instance (inside your Application class onCreate() method:

```
provideApplication(this)
```

REGISTRATION

To create a secure user token, serving to authenticate a user, you will need to use the registration manager. Its only responsibility is to register a user account, containing all the necessary information. Once you have properly configured its instance singleton, you will be able to register new users.

Procedure

To setup the registration manager, do the following:

1. Instantiate the manager:

```
val manager = NoPassRegistrationManager.configure()
       .setOnConfirmationCodeRequiredListener {code ->
           //show it to a user
       }
       .setOnRegistrationResultListener { success, error/*null if no error
occurred*/ ->
          //react to the registration result.
        }
       .build()
```

2. You will need the parameters string from a QR-code, deep link or whatever way of delivering this data to the app you choose. Once you get it, you can trigger the registration process:

```
manager.register(paramsString)
```

Note: NoPass does not deliver any of the user data to outer networks since it is connected to your organization’s on-premises backend system.

AUTHORIZATION

Once you have registered the user, your NoPass backend server gets able to send authorization requests. To receive them, perform the following setup.

The `NoPassUserManager` class serves to receive authorization requests and send authorization responses.

To be able to receive and process requests do the following:

```
val userManager = NoPassUserManager.configure
    .setOnAuthRequestReceivedListener
 { noPassUser, keyphraseData, initialProgress ->
        //noPassUser contains the information about the user
        //keyphraseData contains picture and digits for showing to the user
        //initialProgress approximately shows how much time has passed since the
request //was sent
        //TODO: parse the user data to notify the user
}
    .setOnOTPChangedListener { keyphraseData ->
    //TODO: use the following data to show to the user inside the authentication
message
    //the timeToLive property stands for the time in seconds the OTP and image
values are
    //valid. After that time the callback will be invoked once again
}
  .setOnTickListener{ tick ->
    //current timer tick in seconds
}
    .setOnAuthenticationResultListener{success, error ->
    //TODO: process the server's response on auth request
}
.setOnSessionTimeoutListener{user ->
    //the session is timed out. You can no longer authenticate during this session
}
    .build()
```

When the setup is ready, you can authorize your user during the authentication session by calling the following function:

```
userManager.authenticate()
```

or decline the authentication if needed:

```
userManager.decline(reason)
```

OTHER OPERATIONS

There are other operations that can be performed with the user manager.

1. Get all the users list:

```
userManager.getAllUsers { users, error ->

}
```

2. Get the authentication history:

```
userManager.getAuthenticationHistory { records, error ->

}
```

3. You can also delete the user from both backend and the mobile device:

```
userManager.deleteUser(params.id) {user, success, error ->
        //in case it is triggered from the app the result may be
        //processed in this callback
}

.setOnUserDeleteResultListener { noPassUser, success, throwable ->
    //react to the result. May be useful for "silent" deletions
}
```

4. User updating. This can occur only when initiated from the backend.

```
.setOnUserUpdateResultListener { noPassUser, success, throwable ->
    //react to the result
}
```

5. Backup. It is also possible to create a backup data string encrypted by a 6-digits pin-code. This data may be used to restore accounts later. To initiate backup, do the following:

```
userManager.backupAccounts(pin)

.setOnUsersBackupResultListener { backup,
                        usersList,
                        errors,
                        generalError ->
    //backup string is to be saved somewhere to preserve
}
```

6. Restore. If there are properly backed up accounts, they can be restored by a backup string and the pin code. To begin restoration, run the command:

```
userManager.restoreAccounts(backupData, pin)

.setOnUsersRestoreResultListener { allSuccess,
                        usersList,
                        errors,
                        generalError ->
}
```