ℹ️Introduction

Before enabling A2A integration to your solution you should be already admitted into the **Nautilus Portal** system and have registration of your company, merchant and terminals, if you do not have any registration, then fill in this form please and we will contact you soon as possible.

Before starting of integration you are required to be enrolled as **Integrator** in NFCtron systems. Then you will be given ***token***, which should be part of every A2A API call to Nautilus within A2A. Every instance of terminal should be enabled for integration in Nautilus TMS system.

System requirements:

- Android 9+
- build-in NFC sensor

Architecture requirements:

- attached Android *Service* to payment Android *Activity*

Enter "Integrator program"

Firstly, you need to fill the **Integration request form** on NFCtron Pay site. Afterward, you will be contacted by NFCtron Pay and interviewed for purpose of use and required quantity.

Once you will be contacted by boarding team, you will get access to following chat channels.

- **nautilus_integrator_info** - General annoucements (read-only)
- **nautilus_integrator_implementation** - FAQ channel for issues with integration
- **nautilus_integrator_bug_report** - Reporting of bugs by partners
- **nautilus_integrator_updates** - New versions changelogs and informations about breaking changes

Enable integration for terminal

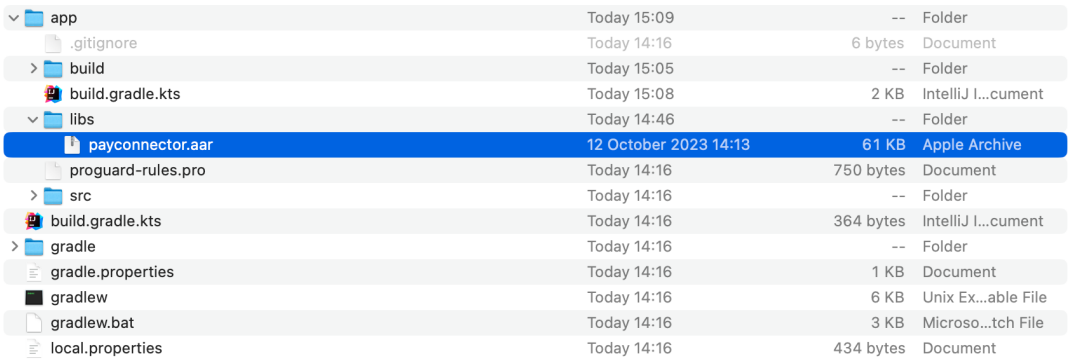
Instructions will be received by email directly from NFCtron Pay support.

🔧Setup

Importing .AAR

This library defines API for integrators. Library contains only predefined interfaces for communication with Nautilus payment app. For more details about API methods, refer to Integration section.

Firstly you need to add `.aar` file into the project. Typically into the `app/libs`.



Typical location of AAR file

Secondly you have to reference to `.aar` package from `app/build.kts.gradle` or `app/build.gradle` using the following snippet.

```
...

dependencies {
    implementation(files("libs/payconnector.aar"))
    ...
```

Attaching to the Service

The first step during integration should be creation of Service, that will be bounded to *Nautilus* **Gateway Service** and capture of `IBinder` (cast to `PayConnectorInterface`) that will be handling all interactions with *Nautilus*.

```
class PayControllerClient : ServiceConnection {
    private val tag = "PayControllerClient"

    // Instance of interface defined in imported .AAR
    var connection: PayConnectorInterface? = null
    var isAlive: Boolean = false

    override fun onServiceConnected(name: ComponentName?, service: IBinder?) {
        Log.d(tag, "Connecting $name")
        try {
            // Cast Binder to predefined interface in imported .AAR
            connection = PayConnectorInterface.Stub.asInterface(service)
            Log.d(tag, "Connected to $name")
            isAlive = true
        } catch (e: Exception) {
            Log.d(tag, "Unable to connect to $name")
        }
    }

    override fun onServiceDisconnected(name: ComponentName?) {
        Log.d(tag, "Disconnected from $name")
        isAlive = false
    }

    override fun onNullBinding(name: ComponentName?) {
        Log.d(tag, "Disconnected from $name, cause null binding")
        isAlive = false
    }

    override fun onBindingDied(name: ComponentName?) {
        Log.d(tag, "Disconnected from $name, cause died binding")
        isAlive = false
    }
}
```

Secondly, you need to bind the `PayControllerClient` to the **Gateway Service** using the following snippet. The binding should be done in awareness of the Android lifecycle (`onCreate`) and after losing connection to the gateway service, which can happen during the runtime of your application and is indicated by the `isAlive` variable in the previous snippet.

```
val intent = Intent()
intent.component = ComponentName(
    "com.nfctron.nautilus",
    "com.nfctron.service.gateway.GatewayService"
)
val connected = applicationContext?.bindService(
    intent, this, Context.BIND_AUTO_CREATE
)
```

Status of connection could be observed from the resulting `boolean` value of `bindService(...)`.

Troubleshooting

- Call on `bindService(...)` keeps returning `false`, but Nautilus app is installed.
  - Verify typos in code
  - Contact us in **nautilus_integrator_implementation** with your *minimal reproducible example code* and provide us a description of running environment.

Testing connection

To verify whether your instance of `PayConnectorInterface` is working, you should call **`getStatusIntegration`**. See this section for more details about method.

Documentation

Operation	Description
Register integration	**(Mandatory)** Registration of 3rd party app for communication. Without proper call on this procedure, other authorized procedures will not be allowed.
Prepare transaction	**(Mandatory)** Each payment should be prepared before it's execution, this provides ability to track payments in real-time and preprocess much as possible before transaction execution. Should be called just before transaction execution.
Execute transaction	**(Mandatory)** Invocation of prepared transaction, should be called just after transaction preparation.
Get info about transaction	**(Optional)** Finding data about a previous transaction on the same device only searches cached data.
Get integration status	**(Optional)** Serves state of integration registration for given token.
Unregister integration	**(Optional)** Disables access for authorized procedures for given token.

Register integration

Before you will be capable to interact with *Nautilus* **Gateway Service,** you have to register connection for first use and define allowed apps to communicate with given token.

Repeative calls with valid token will be blocked and `OPERATION_ERROR` status will be returned. Just one registation of token for each device is allowed.

If you will reinstall the app, you have to register your app once again.

Implementation example

```
val connector: PayConnectorInterface = ...

connector.registerIntegration(RegisterIntegrationRequestModel().apply {
    this.integrationToken = token
    // Application ID of your app(s)
    allowedPackageNames = arrayOf("com.nfctron.sampleintegratorapp")
}, object : RegisterIntegrationResponseListener.Stub() {
    override fun onSuccess() {
        // Do something on success
    }

    override fun onFailure(error: Int) {
        // Do something on error (show on UI, report to your server solution)
    }
})
```

AIDL Definition

```
oneway void registerIntegration(in RegisterIntegrationRequestModel request, in
RegisterIntegrationResponseListener listener);

parcelable RegisterIntegrationRequestModel {
    String integrationToken;
    String[] allowedPackageNames;
}

interface RegisterIntegrationResponseListener {
    void onSuccess();
    void onFailure(int errorCode);
}
```

Troubleshooting

- Why I'm receiving `NOT_FOUND_INTEGRATION_ERROR` ?
    - Your token may be invalid or application do not have the newest data from server, consider to restart app.
- Why I'm receiving `NOT_ACTIVE_INTEGRATION_ERROR` ?
    - Your integration may be disabled in *Nautilus Portal*.

Prepare transaction

Preparation of transaction is nessacary part of transaction execution. By preparation you will capture transaction lifetime in real-time.

Procedure should be called just before transaction execution, keep in mind that preparation may take up to 500ms.

Implementation example

```
val connector: PayConnectorInterface = ...

connector.prepareTransaction(PrepareTransactionRequestModel().apply {
    this.integrationToken = token
    this.transactionId = UUID.randomUUID().toString()
    this.amount = 1200
    this.transactionType = 1
    this.currencyIsoCode = "CZK"
}, object : PrepareTransactionResponseListener.Stub() {
    override fun onSuccess(transactionId: String) {
        // Do something or call directly transaction execution
        connector.executeTransaction(ExecuteTransactionRequestModel().apply {
            this.integrationToken = token
            this.transactionId = transactionId
        }, refListener)
    }

    override fun onFailure(error: Int) {
        // Do something on error (show on UI, report to your server solution)
    }
})
```

AIDL Definition

```
oneway void prepareTransaction(in PrepareTransactionRequestModel request, in
PrepareTransactionResponseListener listener);

parcelable PrepareTransactionRequestModel {
    String transactionId;
    int amount;
    String currencyIsoCode;
    int transactionType;
    @nullable IntNullable externalId;

    String integrationToken;
}

interface PrepareTransactionResponseListener {
    void onSuccess(String transactionId);
    void onFailure(int errorCode);
}
```

Execute transaction

After preparing, you are eligible to execute transactions. All the necessary information will be provided upon preparation.

If your application closes during the execution of a transaction, you can call Get info about transaction to check the status of the transaction.

Implementation example

```
val connector: PayConnectorInterface = ...

connector.executeTransaction(ExecuteTransactionRequestModel().apply {
    this.integrationToken = token
    this.transactionId = transactionId
}, object : ExecuteTransactionResponseListener.Stub() {
    override fun onSuccess(result: ExecuteTransactionResponseModel) {
        // Do something on success
        val state = when (result?.transactionState) {
            TransactionStateEnum.INITIAL -> "INITIAL"
            TransactionStateEnum.ABORTED -> "ABORTED"
            TransactionStateEnum.APPROVED -> "APPROVED"
            TransactionStateEnum.DECLINED -> "DECLINED"
            TransactionStateEnum.IN_PROGRESS -> "IN PROGRESS"
            else -> "Unknown value"
        }
    }

    override fun onFailure(error: Int) {
        // Do something on error (show on UI, report to your server solution)
    }
})
```

AIDL Definition

```
oneway void executeTransaction(in ExecuteTransactionRequestModel request, in
ExecuteTransactionResponseListener listener);

parcelable ExecuteTransactionRequestModel {
    String transactionId;
    String integrationToken;
}

parcelable ExecuteTransactionResponseModel {
    String transactionId;
    int transactionState;
    String responseCode;
    String authorizationCode;
    String referenceNumber;
    String receiptUrl;
}

interface ExecuteTransactionResponseListener {
    void onSuccess(inout ExecuteTransactionResponseModel response);
    void onFailure(int errorCode);
}
```

Get info about transaction

Obtaining transaction details can be useful if your app does not receive a result from the transaction execution. This call can only return cached transactions.

Implementation example

```
val connector: PayConnectorInterface = ...

val transactionId = ...

connector.getInfoTransaction(GetInfoTransactionResponseListener().apply {
    this.integrationToken = token
    this.transactionId = transactionId
}, object : GetInfoTransactionResponseModel.Stub() {
    override fun onSuccess(result: GetInfoTransactionResponseModel) {
        // Do something on success
    }

    override fun onFailure(error: Int) {
        // Do something on error (show on UI, report to your server solution)
    }
})
```

AIDL Definition

```
oneway void getInfoTransaction(in GetInfoTransactionRequestModel request, in
GetInfoTransactionResponseListener listener);

parcelable GetInfoTransactionRequestModel {
    String transactionId;
    String integrationToken;
}

parcelable GetInfoTransactionResponseModel {
    String transactionId;
    int transactionState;
    String responseCode;
    String authorizationCode;
    String referenceNumber;
    String receiptUrl;
}

interface GetInfoTransactionResponseListener {
    void onSuccess(inout GetInfoTransactionResponseModel response);
    void onFailure(int errorCode);
}
```

Get status

If you are unsure whether registration has occurred, ask for the status.

This procedure can also be used as a connection check for *NFCtron Nautilus*.

Implementation example

```
val connector: PayConnectorInterface = ...

connector.getStatusIntegration(GetStatusIntegrationRequestModel().apply {
    this.integrationToken = token
}, object : GetStatusIntegrationResponseListener.Stub() {
    override fun onSuccess(result: GetStatusIntegrationResponseModel) {

    }

    override fun onFailure(error: Int) {

    }
})
```

AIDL Definition

```
oneway void getStatusIntegration(in GetStatusIntegrationRequestModel request, in
GetStatusIntegrationResponseListener listener);

parcelable GetStatusIntegrationRequestModel {
    String integrationToken;
}

parcelable GetStatusIntegrationResponseModel {
    String[] allowedPackageNames;
    int integrationStatus;
}

interface GetStatusIntegrationResponseListener {
    void onSuccess(inout GetStatusIntegrationResponseModel response);
    void onFailure(int errorCode);
}
```

Unregister integration

In scenarios where you want to remove your app from a device, you can call the unregister procedure and disable further calls using the provided token.

Implementation example

```
val connector: PayConnectorInterface = ...

connector.unregisterIntegration(UnregisterIntegrationRequestModel().apply {
    this.integrationToken = token
}, object : UnregisterIntegrationResponseListener.Stub() {
    override fun onSuccess() {
        // Do something on success
    }

    override fun onFailure(error: Int) {
        // Do something on error (show on UI, report to your server solution)
    }
})
```

AIDL Definition

```
oneway void unregisterIntegration(in UnregisterIntegrationRequestModel request, in
UnregisterIntegrationResponseListener listener);

parcelable UnregisterIntegrationRequestModel {
    String integrationId;
    String integrationToken;
}

interface UnregisterIntegrationResponseListener {
    void onSuccess();
    void onFailure(int errorCode);
}
```

💬Changelog

Date	Changes	Description
13.2.2024	Whole document with assests.	Intitial version of intrgration documentation.

🗃️ Assets

Will be provided during communication with support of NFCtron Pay.

📖Dictionary

Term	Meaning
A2A	"A2A" stands for "App-to-App" and refers to a type of integration where two separate software applications or mobile apps communicate directly with each other, typically using APIs (Application Programming Interfaces). This direct communication allows data and functionality to be shared between the apps, enhancing user experiences and enabling new features. A2A integration is commonly used in various contexts, including payment processing, data sharing, and cross-platform functionality.
AIDL	"AIDL" stands for "Android Interface Definition Language". It's a special language used in Android development for defining the interface requirements between client and service in a remote service. In simpler terms, it's a way for different parts of an Android app to communicate with each other, especially when they are running in different processes or even on different devices. AIDL helps in defining the methods that can be called remotely, allowing components to interact seamlessly.
AAR	"AAR" in the context of Android typically refers to "Android Archive". An AAR file is a binary distribution of an Android Library Project. It contains resources such as compiled code (JAR files), resources (assets, images), and a manifest file. AAR files are used for sharing code and resources across different Android projects, providing a convenient way to encapsulate and distribute reusable components.
TMS	"TMS" stands for "Transaction Management System" in the context of payment systems. It's a software solution used by businesses to manage and process transactions, typically including tasks such as authorizing payments, processing refunds, generating reports, and reconciling transactions. TMS systems help streamline the payment process, improve efficiency, and ensure accuracy in financial transactions. They are commonly used by merchants, banks, and other financial institutions to handle electronic transactions securely and efficiently.