Geidea Gateway Android SDK Integration Guide

**Version 2.0**

|  |  |
| --- | --- |
| Document version | 2.0 |
| Date of version | 06.01.2022 |
| Author | Svetlozar Kostadinov |

|  |
| --- |
|  |

Version History

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Changes |
| 1.0 | 24.03.2021 | Svetlozar Kostadinov | Initial version |
| 2.0 | 06.01.2022 | Svetlozar Kostadinov | 3DS v2, Payment Intents, e-Invoices, Meeza QR, Arabic localization |

Contents

[Introduction 4](#_Toc67574686)

[1.1 Document Purpose 4](#_Toc67574687)

[Android SDK Integration 5](#_Toc67574688)

[1.2 Requirements 5](#_Toc67574689)

[1.3 How to start 5](#_Toc67574690)

[1.3.1 Gradle setup 5](#_Toc67574691)

[1.3.2 SDK Initialization 5](#_Toc67574692)

[1.4 Getting your Merchant configuration 6](#_Toc67574693)

[1.5 Integration types overview 6](#_Toc67574694)

[1.6 The “UI Flow” concept 6](#_Toc67574695)

[1.7 Payment flow 7](#_Toc67574696)

[1.7.1 Building your PaymentIntent 7](#_Toc67574697)

[1.7.2 Starting Payment flow 8](#_Toc67574698)

[1.7.3 Receiving the Order result 9](#_Toc67574699)

[1.7.4 PaymentIntent properties specification 9](#_Toc67574700)

[1.7.5 PaymentMethod properties specification 12](#_Toc67574701)

[1.8 API 13](#_Toc67574702)

[1.8.1 GeideaPaymentAPI 13](#_Toc67574703)

[1.9 UI Components 14](#_Toc67574704)

[1.9.1 Introduction 14](#_Toc67574705)

[1.9.2 Types 14](#_Toc67574706)

[1.9.3 Validation of forms 14](#_Toc67574707)

[1.9.4 Validation of input fields 15](#_Toc67574708)

[1.9.5 Styling 16](#_Toc67574709)

[1.9.6 Simple payment form with PaymentFormView 17](#_Toc67574710)

[1.9.7 Custom payment form by composing Geidea components 21](#_Toc67574711)

[1.9.7.1 CardInputView 21](#_Toc67574712)

[1.9.7.2 AddressInputView 23](#_Toc67574713)

[1.9.8 Custom payment form with non-Geidea components 25](#_Toc67574714)

[1.9.9 Custom integration 25](#_Toc67574715)

[1.10 Code samples 26](#_Toc67574716)

[1.11 Response codes 27](#_Toc67574717)

[1.11.1 Response codes and messages 27](#_Toc67574718)

[1.11.2 Detailed response code and message groups: 27](#_Toc67574719)

[Appendix A. Glossary 33](#_Toc67574720)

# Introduction

## Document Purpose

The purpose of this Integration guide is to serve as a technical documentation for merchants who wish to integrate Geidea Payment SDK for Android so that they can use Payment Gateway services in their Android application.

This guide describes the functionality and APIs provided by the SDK and different approaches to integrate the SDK and customize it.

# Android SDK Integration

## Requirements

* Android 6.0+
* Kotlin or Java 8

## How to start

### Gradle setup

The SDK is distributed as an archive containing Maven AAR artifact and POM file. The easiest way to import it into your project is to:

1. Unpack the SDK archive with the AAR and POM files into some local folder (may not necessarily be under your project directory)
2. Add it as a local Maven repository to your project-level build.gradle

**Example**

allprojects {

repositories {

maven { url "<LOCAL PATH TO THE EXTRACTED AAR AND POM>" }

}

}

3. Add the SDK as a dependency in your app-level build.gradle

**Example**

implementation 'net.geidea.paymentsdk:paymentsdk:<LATEST VERSION>'

Now Geidea Payment SDK should be imported in your project.

### SDK Initialization

As an initialization step the SDK expects that you provide your Merchant credentials with the GeideaPaymentAPI.setCredentials() method. However, it is not required to set them on each start. It could be once per installation of the app as the credentials are persisted securely encrypted on device. You can check if there are credentials already stored with the GeideaPaymentAPI.hasCredentials property. It is important that they stored before using the SDK.

**Example**

if (!GeideaPaymentAPI.hasCredentials) {

GeideaPaymentAPI.setCredentials(

merchantKey = "<YOUR MERCHANT KEY>",

merchantPassword = "<YOUR MERCHANT PASSWORD>"

)

}

**IMPORTANT**  
Do not hard-code your merchant password directly into your APK. Instead get it dynamically (from an endpoint of your backend or elsewhere) due to security reasons.

## Getting your Merchant configuration

After the SDK initialization and before performing transactions it might be useful to obtain your current Merchant configuration.

**Kotlin**

Use GeideaPaymentAPI.getMerchantConfiguration() suspend function. Note that you do not need to call it inside IO context because this is done automatically by the SDK.

coroutineScope.launch(context = Dispatchers.Main) **{** // Automatically called on Dispatchers.IO context  
 val configuration = GeideaPaymentAPI.getMerchantConfiguration()  
**}**

**Java**

GeideaPaymentAPI.getMerchantConfiguration(result -> {  
 // Handle success/error result  
});

## Integration types overview

There are few different approaches to integrate Geidea payment services into your app. The choice what approach to follow depends mainly on your needs for UI customizability.

1. **Simple** integration – SDK hosts the entire UI flow and performs all transactions. A “turnkey” solution that requires minimal setup. You simply call a method to start the Payment flow and then receive your Order after everything is ready. For more info refer to [“UI Flow” concept](https://endava-my.sharepoint.com/personal/svetlozar_kostadinov_endava_com/Documents/Geidea/Releases/deliverable-Jan6-2022/UI_Flow#_) and [Payment flow](#_Payment_flow).
2. **Mixed** integration – your Merchant app hosts the payment form UI. SDK performs the authentication and all transactions.
3. **Custom** integration – the Merchant app hosts entire UI flow (payment form, authentication) and performs all transactions by calling the Direct [API](#_API).

## The “UI Flow” concept

SDK uses the concept of UI flow which is a sequence of UI screens, network calls and various other operations all encapsulated in one Activity call for result. UI flows are implemented based on the typical Android Activity results where one SDK activity (or a chain of more than one) is launched with an **input** intent, then it performs its work and finally produces some **output** which contains the result data you are interested in (usually the Order). A flow is represented and managed by an ActivityContract implementation.

**Using Activity result contracts**

Instead of relying on the traditional and now deprecated startActivityForResult() method the SDK embraces the newer Activity Result APIs which offer some benefits for you. For more info please visit <https://developer.android.com/training/basics/intents/result>.

## Payment flow

The Payment flow expects an input of type PaymentIntent and returns a result of type GeideaResult<Order> at the end of the payment process. The PaymentContract is used to manage the input/output parcelization.

Declare a launcher somewhere in your code from where you wish to start the payment…

**private var paymentLauncher**: ActivityResultLauncher<PaymentIntent>

…and then register it with a PaymentContract instance and your function or lambda that should accept the final result.

**fun** handleOrderResult(result: GeideaResult<Order>) {

*/\*\* Handle the order response here \*/*

}

**paymentLauncher** = registerForActivityResult(PaymentContract(), ::handleOrderResult)

### Building your PaymentIntent

PaymentIntent contains details about the order, customer and preferred payment method. It has two mandatory properties – amount and currency.

**Kotlin**

**val paymentIntent** = *PaymentIntent* **{** *// Mandatory properties* **amount** = 123.45  
 **currency** = **"SAR"  
 paymentMethod** = PaymentMethod **{** cardHolderName = **"John Doe"** cardNumber = **"5123450000000008"** expiryDate = ExpiryDate(month = 1, year = 25)  
 cvv = **"123"  
 }** *// Optional properties* **callbackUrl** = **"https://website.hook/"  
 merchantReferenceId** = **"1234"  
 customerEmail** = **"email@test.com"  
 billingAddress** = Address(  
 countryCode = **"SAU"**,  
 city = **"Riyadh"**,  
 street = **"Street 1"**,  
 postCode = **"1000"** )  
 **shippingAddress** = Address(  
 countryCode = **"SAU"**,  
 city = **"Riyadh"**,  
 street = **"Street 1"**,  
 postCode = **"1000"** )  
**}**

**Java**

PaymentIntent paymentIntent = **new** PaymentIntent.Builder()  
 .setAmount(123.45d)  
 .setCurrency(**"SAR"**)  
 .setPaymentMethod(**new** PaymentMethod.Builder()  
 .setCardHolderName(**"John Doe"**)  
 .setCardNumber(**"5123450000000008"**)  
 .setExpiryDate(**new** ExpiryDate(1, 25))  
 .setCvv(**"123"**)  
 .build()  
 )  
 .setCallbackUrl(**"https://website.hook/"**)  
 .setMerchantReferenceId(**"1234"**)  
 .setCustomerEmail(**"email@test.com"**)  
 .setBillingAddress(**new** Address(  
 **"SAU"**,  
 **"Riyadh"**,  
 **"Street 1",**  
 **"1000"** ))  
 .setShippingAddress(**new** Address(  
 **"SAU"**,  
 **"Riyadh"**,

**"Street 1",**  
 **"1000"** ))  
 .build();

**Card data input**

The Payment flow supports two ways of collecting the card data:

1. **By the SDK** – when the paymentMethod property is not set or set to null. In this case the SDK will display a hosted native payment form to collect the card and address data.
2. **By the Merchant** – when you supply paymentMethod value. In this case you are responsible to collect the card data through your app UI and hand it to the SDK. The SDK will not show a payment form but will directly perform the necessary transactions and authentications.

**Validations**

Multiple basic validation checks are performed on construction of PaymentIntent and PaymentMethod. E.g. check if the CVV is 3 or 4 digits. If some validation is not satisfied an IllegalArgumentException with a message is thrown. For a comprehensive list of validity conditions please refer to Section 3.6. The full validation is performed server-side and FieldValidationError is returned on bad input.

### Starting Payment flow

Example:

paymentLauncher.launch(paymentIntent)

### Receiving the Order result

The final result of the Payment flow is returned as a sealed object of type GeideaResult<Order>.

**Example**

**fun** handleOrderResult(result: GeideaResult<Order>) {  
 **when** (result) {  
 **is** GeideaResult.Success<Order> -> {  
 *// Payment successful, order returned in result.data* }  
 **is** GeideaResult.Error -> {  
 **when** (result) {  
 **is** GeideaResult.FieldValidationError -> {  
 *// Client error - invalid field values (e.g. a CVV with letters)* handleFieldValidationError(  
 result.type,  
 result.title,  
 result.status,  
 result.traceId,  
 result.errors  
 )  
 }  
 **is** GeideaResult.NetworkError -> {  
 *// Client or server error* handleNetworkError(  
 result.responseCode  
 result.responseMessage  
 result.detailedResponseCode  
 result.detailedResponseMessage  
 )  
 }  
 **is** GeideaResult.SdkError -> {  
 *// An unexpected error due to improper SDK  
 // integration or SDK internal bug* handleSdkError(result.errorCode, result.errorMessage)  
 }  
 }  
 }  
 **is** GeideaResult.Cancelled -> {  
 *// Payment flow cancelled by the user (e.g. Back button)*   
 }  
 }  
}

### PaymentIntent properties specification

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parameter name | Description | Type | Validations | Optionality |
| amount | Transaction amount | BigDecimal | Must be decimal, positive and must not more than 2 digits after decimal point | Mandatory |
| currency | Currency of the amount, standard:  ISO 4712 currency code | String | Must be 3 characters long | Mandatory |
| paymentOperation | If null the effective value is read from merchant configuration | String | Possible values:   * null, * “Pay” * “PreAuthorize” * “AuthorizeCapture” | Optional |
| initiatedBy |  |  | Possible values:   * “Internet”, * “Merchant” | ? |
| callbackURL | The response with order details, will be returned to this URL | String | Must be a valid URL and to have an HTTPS protocol | Optional |
| merchantReferenceId | Use this as your unique reference for each transaction | String | \* | Optional |
| customerEmail | Customer email if you already have it | String | Must be a valid email address | Optional |
| billingAddress  For billing address – the following parameters can be passed:   * countryCode * street * city * postCode | Billing address details for thecustomer if you already have it  country standard:  ISO 3166 – alpha-3 code | String | **countryCode -** must be 3 characters  **street, city, postcode –** max 255 characters | Optional |
| shippingAddress  For shipping address – the same parameters as for billing can be passed:   * countryCode * street * city * postCode | Shipping address details for thecustomer if you already have it  country standard:  ISO 3166 – alpha-3 code | String | **countryCode -** must be 3 characters  **street, city, postcode –** max 255 characters | Optional |
| cardOnFile | Allow/disallow tokenization of the payment card (optional). Default is false. | Boolean | Should be set to true only if the tokenization is enabled in your Merchant configuration. | Optional |
| agreementType | Type of agreement between Merchant and customer on future use of a tokenized card | String | Possible values:   * null, * “Recurring”, * “Installment”, * “Unscheduled” | Mandatory if agreementId is not null |
| agreementId | Agreement ID | String |  | Mandatory if agreementType is not null |
| eInvoiceId | e-Invoice ID | String | e-Invoice can be created with GeideaPaymentAPI.createEInvoice() | Optional |

\*Please note that all above parameters have a max length of 255 symbols.

### PaymentMethod properties specification

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parameter name | Description | Type | Client-side Validations | Optionality |
| cardHolderName | The name appearing on the physical card | String | 5..255 length | Mandatory |
| cardNumber | Card PAN | String | Non-empty | Mandatory |
| expiryDate | Card expiry date | ExpiryDate | - | Mandatory |
| expiryDate.month | Card expiry month | Int | 1..12 | Mandatory |
| expiryDate.year | Card expiry year | Int | 1..99 | Mandatory |
| owner |  | String | - | Optional |

## API

### GeideaPaymentAPI

**Managing your Merchant credentials**

fun setCredentials(merchantKey: String, merchantPassword: String)

fun hasCredentials: Boolean

fun clearCredentials()

**Choosing between Live / Test environment**

fun getServerEnvironment(): ServerEnvironment

fun setServerEnvironment(serverEnvironment: ServerEnvironment)

**Getting your merchant configuration**

suspend fun getMerchantConfiguration(): GeideaResult<MerchantConfigurationResponse>

**Transactions with Direct API**

suspend fun authenticate(authenticationRequest: AuthenticationRequest): GeideaResult<AuthenticationResponse>

suspend fun pay(paymentRequest: PaymentRequest): GeideaResult<Order>

suspend fun payWithToken(tokenPaymentRequest: TokenPaymentRequest): GeideaResult<Order>

suspend fun capture(captureRequest: CaptureRequest): GeideaResult<Order>

suspend fun cancelOrder(cancelRequest: CancelRequest): GeideaResult<PaymentResponse>

**e-Invoice API**

suspend fun createEInvoice(createEInvoiceRequest:CreateEInvoiceRequest):GeideaResult<EInvoiceResponse>

suspend fun getEInvoice(eInvoiceId: String): GeideaResult<EInvoiceResponse>

suspend fun updateEInvoice(updateEInvoiceRequest:UpdateEInvoiceRequest):GeideaResult<EInvoiceResponse>

suspend fun deletePaymentIntent(paymentIntentId: String): GeideaResult<EInvoiceResponse>

**Meeza QR API**

suspend fun createMeezaPaymentQrCode(createPaymentIntentRequest: CreateMeezaPaymentIntentRequest): GeideaResult<MeezaQrImageResponse>

suspend fun sendMeezaRequestToPay(meezaPaymentRequest: MeezaPaymentRequest): GeideaResult<MeezaPaymentResponse>

**Order API**

suspend fun getOrders(orderSearchRequest: OrderSearchRequest): GeideaResult<OrderSearchResponse>

suspend fun getOrder(orderId: String): GeideaResult<Order>

suspend fun captureOrder(captureRequest: CaptureRequest): GeideaResult<Order>

suspend fun refundOrder(refundRequest: RefundRequest): GeideaResult<Order>

suspend fun cancelOrder(cancelRequest: CancelRequest): GeideaResult<PaymentResponse>

**Cleaning up**

fun cleanup()

**Note**: As a full developer reference of the API please use the provided KDoc / Javadoc packages.

## UI Components

### Introduction

Developing your own payment UI is **not** necessary in case of **Simple** integration type. Everything necessary to perform a payment is integrated in the Payment flow and is working out-of-the-box. However, in the **Mixed** integration type you may embed some or the entire UI inside your app. In the **Custom** integration type your app hosts all the UI and performs the necessary transactions. The SDK provides Material components that make creating of your own payment UI for Mixed and Custom integrations easier compared to creating your own UI from scratch. Many daunting or annoying tasks like card data validation are handled for you internally in the components.

### Types

The SDK includes a set of Material components to help you build a modern payment UI. There are two types of components:

* Form views (composite views)
  + PaymentFormView
  + CardInputView
  + AddressInputView
* Input fields – sub-classes of TextInputEditText
  + CardNumberEditText
  + CardHolderEditText
  + CardExpiryEditText
  + CardSecurityCodeEditText
  + EmailEditText
  + CountryAutoCompleteTextView
  + StreetEditText
  + CityEditText
  + PostCodeEditText

### Validation of forms

Validation is handled by the Form views internally. All Form views expose the Boolean val property isValid which is automatically updated in sync with the input. Listening for validation status changes can be done with the following calls:

paymentFormView.setOnValidationChangedListener { value: PaymentFormData?, valid: Boolean ->

}

cardInputView.setCardInputListener(object : CardInputAdapter() {  
 override fun onCardValidationChanged(cardValid: Boolean) {

*// Called when input data becomes valid or invalid*  
 }  
})

addressInputView.setAddressInputListener(object : AddressInputAdapter() {  
 override fun onAddressValidationChanged(address: Address, valid: Boolean) {

*// Called when input data becomes valid or invalid*  
 }  
})

### Validation of input fields

Geidea Material components delegate the validation of their input to an instance of the Validator interface. At construction the components do not have an attached validator, so assigning an appropriate one with the setValidator() method is responsibility of the higher level component or activity which owns that component.

**Validators**

Validator is a very simple generic interface having a validate() whose only purpose is to check if the current value in the input field is valid or not. If not it returns information for the exact reason why it is invalid. Validators are called on each input change.

**Validation status**

The UI components expose the validation status of their current input as the property

val validationStatus: ValidationStatus

When you are not interested in detailed validation status there is a simpler alternative

val isValid: Boolean

The validationStatus property is updated automatically by the component with the status returned by the associated validator. ValidationStatus is a sealed class with 3 possible sub-types:

* Valid - signifies that the input is acceptable for further processing.

object Valid : ValidationStatus()

* Invalid – signifies that the input is an unacceptable for further processing due to some reason which is specified in the reason property.  
  data class Invalid(val reason: InvalidationReason) : ValidationStatus()
* Undefined – special type for internal use which only means that the validation is not yet performed. Validators must never return this type.

object Undefined : ValidationStatus()

**Listening for (in)validation events**

Use the following two methods to register for validation status changes:

textField.setOnValidStatusListener { value: T ->

textField.updateErrorMessage()

}

setOnInvalidStatusListener { value: T?, invalidStatus: ValidationsStatus.Invalid->

textField.updateErrorMessage()

}

In the example above we call updateErrorMessage() for the field to update its error message from the validation status:

* When valid - the error message is cleared.
* When invalid - the error message is set to the one from reason property.

**Invalidation reasons**

The InvalidationReason subtypes represent a concrete validation errors. Its method getMessage() returns some localized user-friendly message with a short explanation for a concrete validation error. E.g. the message could be set as an inline validation error with TextInputLayout.setError().

**Displaying invalidation reasons**

The validatable input fields do update their validation status on every input event, however the owner of the input fields must explicitly visualize the status. This can be done with the help of the OnErrorListener.

textField.setOnErrorListener { errorMessage: CharSequence? ->

// *Update the parent text input layout with the error message (null when valid)*

textInputLayout.*error* = errorMessage

}

**List of default validators**

For each input field class there is a validator class that should be assigned to the corresponding input field after layout inflation:

|  |  |
| --- | --- |
| **Input field** | **Default validator** |
| CardNumberEditText | DefaultCardNumberValidator |
| CardExpiryEditText | ExpiryDateValidator |
| CardSecurityCodeEditText | CvvValidator |
| CardHolderEditText | CardHolderValidator |
| CountryAutoCompleteTextView | DefaultCountryValidator |
| StreetEditText | FieldMaxLengthValidator |
| CityEditText |
| PostCodeEditText |

### Styling

Geidea Material components are normal Material components that can be styled with the standard approaches:

* through styles defined in an app-level or activity-level theme;
* through directly applying a style to the view definition in your XML;
* by supplying attributes programmatically through the constructor.

For more details please take a look at the [Code samples](#_Code_samples) > SampleThemeActivity.

### Simple payment form with PaymentFormView

PaymentFormView is a complete payment form component or more concretely - a vertical layout which has card and customer data input fields embedded inside and managed by itself.

**Layout hierarchy of PaymentFormView**

The following tree structure represents the public components in the form. Note: only the Geidea components are shown. The form contains more child views like card brand logos and “Pay” button.

* PaymentFormView
  + CardInputView
    - CardNumberEditText
    - CardExpiryDateEditText
    - CardSecurityCodeEditText
    - CardHolderEditText
  + EmailEditText – customer e-mail
  + AddressInputView – billing address
    - CountryAutoCompleteTextView – country drop-down selector
    - StreetEditText
    - CityEditText
    - PostEditText
  + AddressInputView – shipping address with the same child fields as billing address.

**Embedding in your XML layout**

Always wrap the form view in a scrollable container. You may need also a “Pay” or other UI which the customer will use to start the payment flow once the form is populated with valid data.

<ScrollView  
 android:id**="@+id/paymentFormScrollView"** android:layout\_width**="match\_parent"** android:layout\_height**="match\_parent"**>

<LinearLayout

android:layout\_width**="match\_parent"** android:layout\_height**="wrap\_content"**

android:orientation**="vertical"**>

<net.geidea.paymentsdk.ui.widget.PaymentFormView  
 android:id**="@+id/paymentFormView"** android:layout\_width**="match\_parent"** android:layout\_height**="wrap\_content"**/>

<Button

android:id**="@+id/payButton"**

android:layout\_width**="match\_parent"** android:layout\_height**="wrap\_content"**

android:text**="Pay"**/>

<LinearLayout/>

</ScrollView>

**Initializing**

Setup the payment form with your accepted card brands, supported countries, etc.

*paymentFormView.configure(GeideaPaymentAPI.getMerchantConfiguration())*

**Populating**

First, let’s assume we have a val initial: PaymentIntent instance that holds the parameters we will prepopulate the payment form with. Prepopulating the form with it could look like this:

paymentFormView.showCustomerEmail = initial.showCustomerEmail  
if (initial.showCustomerEmail) {  
 paymentFormView.customerEmail = initial.customerEmail  
}  
  
paymentFormView.showAddress = initial.showAddress  
  
initial.billingAddress?.*let* **{** paymentFormView.billingAddress = **it }**initial.shippingAddress?.*let* **{** paymentFormView.shippingAddress = **it }**if (initial.billingAddress?.countryCode.*isNullOrEmpty*()) {  
 paymentFormView.billingAddressCountryCode = **"SAU"**}  
  
if (initial.shippingAddress?.countryCode.*isNullOrEmpty*()) {  
 paymentFormView.shippingAddressCountryCode = **"SAU"**}  
  
val same = paymentFormView.billingAddress.*equalsIgnoreCase*(paymentFormView.shippingAddress)  
paymentFormView.isSameAddressChecked = same || initial.shippingAddress.*isNullOrEmpty*()

**Reading form data and starting the payment flow**

*payButton.setOnClickListener { payButton ->  
 // Create a new “final” payment intent that contains the initial data combined with any data that the customer decided to override.  
 val finalPaymentIntent = PaymentIntent* **{** // Populate the merchant-related properties...  
 paymentOperation = initial.paymentOperation  
 amount = initial.amount  
 currency = initial.currency  
 merchantReferenceId = initial.merchantReferenceId  
 callbackUrl = initial.callbackUrl  
 showCustomerEmail = initial.showCustomerEmail  
 showAddress = initial.showAddress  
 cardOnFile = initial.cardOnFile  
 initiatedBy = initial.initiatedBy  
 agreementId = initial.agreementId  
 agreementType = initial.agreementType  
 bundle = initial.bundle  
  
 // ...then read and populate customer data from the form  
  
 val card: Card? = paymentFormView.card  
 paymentMethod = *PaymentMethod* **{** cardHolderName = card?.cardHolderName  
 cardNumber = card?.cardNumber  
  expiryDate = card?.expiryDate  
 cvv = card?.cvv  
 **}** customerEmail = if (paymentFormView.showCustomerEmail) {  
 paymentFormView.customerEmail  
 } else {  
 initial.customerEmail  
 }  
  
 billingAddress = if (paymentFormView.showAddress) {  
 paymentFormView.billingAddress  
 } else {  
  initial.billingAddress  
 }  
  
 shippingAddress = if (paymentFormView.showAddress) {  
  if (paymentFormView.isSameAddressChecked) {  
  paymentFormView.billingAddress  
 } else {  
  paymentFormView.shippingAddress  
 }  
 } else {  
 initial.shippingAddress  
 }  
 **}***)*

*// Start the Payment flow with the “final” payment intent*

paymentLauncher.launch(finalPaymentIntent) *}*

**Reading the entire content of the form**

The entire content can be read with val data: PaymentFormData.

**Getting / setting individual field values**

It is possible to modify and obtain values for specific fields only. Any changes will affect the respective input field value even if the field is not currently visible.

var card: Card?

var cardNumber: String?

var cardExpiryDate: String?

var cardSecurityCode: String?

var cardHolder: String?

var customerEmail: String?

var showCustomerEmail: Boolean

var showAddress: Boolean

var billingAddress: Address

var billingAddressCountryCode: String?

var billingAddressStreet: String?

var billingAddressCity: String?

var billingAddressPostCode: String?

var shippingAddress: Address

var shippingAddressCountryCode: String?

var shippingAddressStreet: String?

var shippingAddressCity: String?

var shippingAddressPostCode: String?

var isSameAddressChecked: Boolean

var acceptedCardBrands: Set<CardBrand>

var isValid: Boolean

**Enabling the “Pay” button once the form is validated**

paymentFormView.setOnValidationChangedListener { data: PaymentFormData?, valid: Boolean ->

payButton.*isEnabled* = valid

}

**View state saving / restoration**

PaymentFormView and its child fields preserve their entire state. For code example you may look at the SamplePaymentFormActivity in the Demo app.

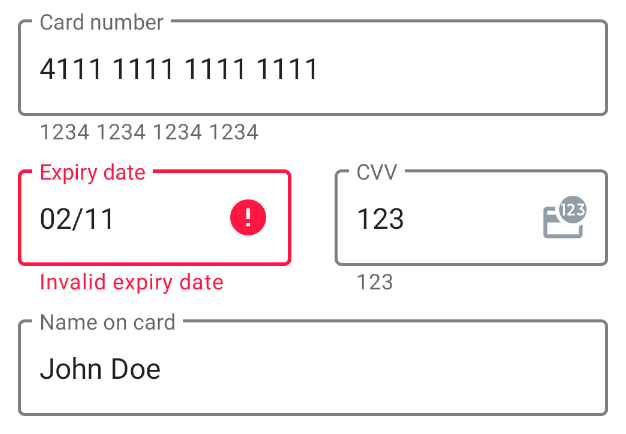
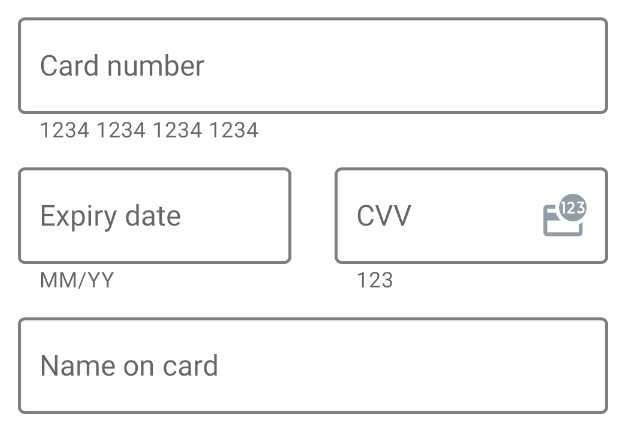
**Advanced use-cases**

If you need more flexibility and customizability than what PaymentFormView offers, then you could use Geidea’s Material components as building blocks. In this way you would have more control over the UI behavior and appearance. For more info see [Custom payment form by composing Geidea components](#_Custom_payment_form).

### Custom payment form by composing Geidea components

#### CardInputView

CardInputView is a layout composed of Material text input fields for the mandatory card data – card holder, number, expiry date and CVV. It’s a self-contained component that does all necessary validation, card brand detection, digits grouping, filtering of invalid characters and restricting to maximal allowed length.



**Embedding in your XML layout**

<net.geidea.paymentsdk.ui.widget.card.CardInputView  
 android:id**="@+id/cardInputView"** android:layout\_width**="match\_parent"** android:layout\_height**="wrap\_content"** />

**Getting the input data**

When the customer enters all required fields with valid input the card property becomes non-null.

val card: Card? = cardInputView.card

**Setting card number**

Partial, separated and non-separated numbers can be set.

cardInputView.setCardNumber(“4111 1111 1111 1111”) // or…

cardInputView.setCardNumber(“4111111111111111”)

The input field will automatically attempt to group the digits according to the grouping convention of the recognized card brand.

**Setting an expiry date**

The expiry date should be in 2-digit MONTH/YEAR format.

cardInputView.setCardExpiryDate(**"12/34"**)

**Setting a security code (CVV)**

The valid security code is 3- or 4-digit code depending on the card brand. It is 4 digits for *American Express* and 3 for all others**.**

cardInputView.setSecurityCode(**"123"**) // or…  
cardInputView.setSecurityCode(**"1234"**)

**Setting a card holder**

The card holder name must have a maximal length of 255 characters.

cardInputView.setCardHolder(**"John Doe"**)

**Checking for validity**

When Customer enters all required fields with valid input the value of isValid property becomes true.

val valid: Boolean = cardInputView.isValid

**Getting the currently recognized card brand**

val cardBrand: CardBrand = cardInputView.cardBrand  
when (cardBrand) {  
 CardBrand.Visa,  
 CardBrand.Mastercard,  
 CardBrand.AmericanExpress,  
 CardBrand.Mada -> {} // Recognized brand  
 CardBrand.Unknown -> {} // Unrecognized or invalid brand  
}

**Filtering only card brands that you accept**

By default, CardInputView accepts all card brands supported by Geidea. It must be configured in order to filter the non-accepted.

val merchantConfig = GeideaPaymentAPI.getMerchantConfiguration()

val acceptedBrands= (merchantConfig.paymentMethods ?: *emptyList*())  
 .*map*(CardBrand::fromBrandName)  
 .*toSet*()  
 .*minus*(CardBrand.Unknown)

cardInputView.setCardBrandFilter(acceptedBrands::contains)

**Setting input listener**

cardInputView.setCardInputListener(object : CardInputView.CardInputAdapter() {  
 override fun onFocusChange(focusField: CardFieldType) {  
 // Implementation is left for sub-classes  
 }  
  
 override fun onFieldValidStatus(field: CardFieldType) {  
 // Implementation is left for sub-classes  
 }  
  
 override fun onFieldInvalidStatus(field: CardFieldType, status: ValidationStatus.Invalid) {  
 // Implementation is left for sub-classes  
 }

override fun onCardValidationChanged(cardValid: Boolean) {  
 payButton.*isEnabled* = cardValid}  
  
 override fun onCardInputComplete() {clearFocusAndDismissKeyboard()}  
})

**Using the card data to pay with**

Here is a minimalistic example of a function that initiates a payment with the collected card data

private fun payWith(card: Card) {

val paymentIntent = *PaymentIntent* **{** amount = **"**123.45**"**.toBigDecimal()  
 currency = **"**SAU**"**

**// Supply the card data to SDK to perform the required authentication / transactions**

paymentMethod = *PaymentMethod* **{** cardNumber = card.cardNumber  
 cardHolderName = card.cardHolderName  
 expiryDate = card.expiryDate  
 cvv = card.cvv  
  **}**

// More *PaymentIntent* parameters could be set here

}  
 paymentLauncher.launch(paymentIntent)

clearFocusAndDismissKeyboard()

}

**Setting card brand listener**

cardInputView.setOnCardBrandChangedListener **{** brand **->**

**// Highlight the icon of the card brand**

**}**

**Setting text watchers**

fun setCardHolderTextWatcher(cardHolderTextWatcher: TextWatcher)  
fun setCardNumberTextWatcher(cardNumberTextWatcher: TextWatcher)  
fun setExpiryDateTextWatcher(expiryDateTextWatcher: TextWatcher)  
fun setSecurityCodeTextWatcher(securityCodeTextWatcher: TextWatcher)

For ease-of-use the abstract TextWatcherAdapter could be extended which has empty implementations of the abstract methods.

**Setting security code (CVV) icon click listener**

fun setSecurityCodeEndIconClickListener(listener: OnClickListener?)

#### AddressInputView

AddressInputView is a layout composed of text fields for entering Address data. It can be used for collecting billing and shipping addresses.

**Embedding in your XML layout**

<net.geidea.paymentsdk.ui.widget.address.AddressInputView  
 android:id**="@+id/addressInputView"** android:layout\_width**="match\_parent"** android:layout\_height**="wrap\_content"** />

**Getting / setting input data**

**The whole address**

var address: Address = addressInputView.address

addressInputView.address = Address { /\*…\*/ }

**Country code**

addressInputView.countryCode = “SAU”

val countryCode: String? = addressInputView.countryCode

**Street**

addressInputView.street = “Wall st. 11”

val street: String? = addressInputView.street

**City**

addressInputView.city = “New York”

val city: String? = addressInputView.city

**Postcode**

addressInputView.postCode = “10005”

val postCode: String? = addressInputView.postCode

**Checking for validity**

When all populated address fields are valid then isValid property becomes true.

val valid: Boolean = addresInputView.isValid

**Setting input events listener**

addresInputView.setAddressInputListener(object : AddressInputAdapter() {

fun onFocusChange(focusField: AddressFieldType) {}  
 fun onFieldValidStatus(field: AddressFieldType) {}  
 fun onFieldInvalidStatus(field: AddressFieldType, status: ValidationStatus.Invalid) {}  
 fun onAddressValidationChanged(address: Address, valid: Boolean) {}

)

### Custom payment form with non-Geidea components

It is possible to use any kind of UI components to collect the payment data. The requirement is to populate a PaymentIntent instance and its paymentMethod property then [launch the Payment flow](#_Payment_flow) with that instance. See SamplePaymentFormActivity in the Demo app.

### Custom integration

Geidea Android SDK provides the possibility for totally seamless integration. This way is intended for complex use-cases and requires significantly more effort that the other integration types (**Simple** and **Mixed)**. Typically, you will need to set-up the UI - a payment form and WebView for authentication, then implement correct transaction flows with the help of the [API](#_API) methods.

## Code samples

Code samples can be found in the Demo app. They demonstrate different types of integration of the SDK:

**SampleSimplestPaymentActivity**

Sample with the minimally required code to perform a payment.

**SampleThemeActivity**

How to customize the Payment flow via customized theme.

**SamplePaymentFormActivity**

How to embed a payment form in your XML layout, populate and read data, listen for events and start payment flow. Uses customized theme.

**SampleAddressActivity**

How to embed an Address input view in your XML, populate and read data, listen for events and manipulate the view.

**SampleAddressFieldsActivity**

How to embed individual Address input fields in your XML, populate and read data, listen for validation events.

**SampleCardActivity**

How to embed an Address input view in your XML, populate and read data, listen for events and manipulate the view.

**SampleCardFieldsActivity**

How to embed individual Card input fields in your XML, populate and read data, listen for validation events.

**SampleEInvoiceActivity**

How to perform CRUD operations with e-Invoices.

**SampleOrdersActivity**

How to get and search by status and date orders. How to get specific order by id.

## Response codes

|  |  |
| --- | --- |
| **Android SDK errors** |  |
| **Response code** | **Response message** |
| 002 | Missing public key |

### Response codes and messages

These response codes and messages are returned to indicate to the user/client what was the result of a payment and if something went wrong (100-800).

|  |
| --- |
| **Response code groups** |
| **Response Code** | **Response Message** | **Description** |
| 000 | Success | Success |
| 100 | General error - Payment was not successful | General error |
| 200 | Payment validation failed | Initial backend validation error (card number, luhn check, etc) |
| 300 | 3DS authentication failed | 3DS errors |
| 400 | Payment authorization failed | Authorize operation errors |
| 500 | Payment capture failed | Capture operation errors |
| 600 | Payment blocked by gateway | Filter service errors |
| 700 | Payment failed | Pay operation errors |
| 750 | Tokenized payment failed | Token errors |
| 800 | Refund failed | Refund operation errors |

### Detailed response code and message groups:

These are the detailed response code and messages in the respective groups that can be seen in the payment response so that merchants understand what has happened in further details with the specific payment and to perform next steps if there has been a failure.

| **Success group (000)** |
| --- |
| **Detailed Response Code** | **Detailed Response Message** |
| 000.000 | Your transaction was successful |
| 000.001 | Your transaction was successful - proceed without 3DS verification code |
| 000.002 | Your transaction was successful - proceed with 3DS verification code |
| 000.003 | Token payment successful - but could not find parentOrderId |

| **General error group (100)** | **(note – HSM from 001-007/others 008-019)** |
| --- | --- |
| **Detailed Response Code** | **Detailed Response Message** |
| 100.001 | HSM invalid input fields |
| 100.002 | Encrypt failed, invalid data |
| 100.003 | Decrypt failed, invalid key |
| 100.004 | Decrypt failed, invalid data |
| 100.005 | Decrypt failed, wrong data-key combination |
| 100.006 | Unable to create key |
| 100.007 | Unable to find key |
| 100.008 | Unable to create order |
| 100.009 | Unable to create transaction |
| 100.010 | Unable to update transaction |
| 100.011 | Order not found |
| 100.012 | Transaction not found |
| 100.013 | Internal Server Error |
| 100.014 | Invalid provider credentials – MPGS |
| 100.015 | MPGS URL not found |
| 100.016 | Invalid provider credentials – GSDK |
| 100.017 | HTTP request failed with connection error |
| 100.018 | Your payment was not successful |
| 100.019 | Unable to update order |
| 100.020 | Cancelled by user |
| 100.021 | Client timed out |
| 100.020 | Cancelled by user |
| 100.021 | Client timed out |
| 100.022 | System timed out |
| 100.023 | Merchant authentication failed |
| 100.024 | Merchant account is not configured for this paymentOperation |
| 100.029 | Unable to disable key |
| 100.020 | Cancelled by user |
| 100.021 | Client timed out |
| 100.030 | Order is already completed |
| 100.031 | Card on file not enabled for this account |

| **Payment validation error group (200)** |
| --- |
| **Detailed Response Code** | **Detailed Response Message** |
| 200.001 | Card is expired |
| 200.002 | Unknown or unsupported card brand |
| 200.003 | Invalid card number length |
| 200.004 | Currency {currency}  is not supported |
| 200.005 | Card brand {cardBrand}  is not supported |
| 200.006 | Invalid card number, Luhn check failed |
| 200.007 | Invalid CVV |

| **3DS group errors (300)** |
| --- |
| **Detailed Response Code** | **Detailed Response Message** | **Description** |
| 300.001 | The card is not 3DS enrolled | Not enrolled (MPGS - do not proceed) |
| 300.002 | The card does not support 3DS | 3DS not supported / card not participating |
| 300.003 | Authentication enrolment check not available | Connection error on enrolment check |
| 300.004 | Customer authentication failed | Failed authentication (MPGS - do not proceed), N response |
| 300.005 | Customer authentication attempted, but could not be completed | M response from authentication (do not proceed) |
| 300.006 | Customer authentication not available | X or U response from MPGS |
| 300.007 | Error parsing authentication response | P response from MPGS |
| 300.008 | Invalid signature on authentication response | S response from MPGS |
| 300.009 | MPI processing error | I response from MPGS |
| 300.010 | Authentication could not be created | Internal service error |
| 300.011 | Invalid JWT | Cybersource specific error when JWT is not valid |
| 300.012 | Invalid Payer Authentication response | Cybersource specific error: PARes status: Missing |
| 300.013 | Attempting processing | Cybersource specific error: PARes status: A |
| 300.014 | Incomplete Authentication | Cybersource specific error: PARes status: U |
| 300.015 | Unsuccessful Authentication | Cybersource specific error: PARes status: N |

| **Authorize operation error group (400)** |
| --- |
| **Detailed Response Code** | **Detailed Response Message** |
| 400.001 | Authorization failed |

| **Capture operation error group (500)** |
| --- |
| **Detailed Response Code** | **Detailed Response Message** | **Description** |
| 500.001 | Capture failed | when MPGS fails |
| 500.002 | Capture not allowed | when orderStatus is not Authorized |
| 500.003 | Invalid order id | when valid order, but different credentials used |

| **Filter service error group (600)** |
| --- |
| **Detailed Response Code** | **Detailed Response Message** |
| 600.001 | Blocked PAN |
| 600.002 | Blocked card BIN |
| 600.003 | Blocked IP country |
| 600.004 | Blocked customer country |

| **Pay operation error group (700)** |
| --- |
| **Detailed Response Code** | **Detailed Response Message** |
| 700.001 | Payment failed |

| **Token error groups (750)** |
| --- |
| **Detailed Response Code** | **Detailed Response Message** |
| 750.001 | Invalid input fields |
| 750.002 | Token not found |
| 750.003 | Token Owner not found |
| 750.004 | System not found |
| 750.005 | AgreementID does not match initial value |
| 750.006 | Parameter initiatedBy not configured for this account |

| **Refund operation error group (800)** |
| --- |
| **Detailed Response Code** | **Detailed Response Message** | **Description** |
| 800.001 | Refund failed | when MPGS fails |
| 800.002 | Refund failed - Order does not exist | when order cannot be found |
| 800.003 | Refund failed - Payment for order has not been completed | when order has not been paid / captured |
| 800.004 | Refund failed - Invalid order | when a merchant tries to refund another merchants Paid/Captured order |

Appendix A. Glossary

|  |  |
| --- | --- |
| Glossary Item | Description |
| SDK | Software Development Kit |
| 3DS | 3D Secure - secure protocol designed to ensure enhanced security and strong authentication |
| MPGS | Mastercard Payment Gateway Services |
| BIN | Bank Identification Number - the initial four to six numbers that appear on a credit/debit card |
| HPP | Geidea Hosted Payment Page |