Manual de Integración del SDK

SDK de Pagos de QRs

Versión: *1.0.1*

Aplicación: *Android e iOS nativo con React Native*

Fecha: *Abril 2025*

Índice

Generar las carpetas de las plataformas nativas

Si tu proyecto de React Native no tiene la carpeta `android` e `ios`, sigue estos pasos:

1. Abre una terminal en la raíz de tu proyecto.
2. Ejecuta el siguiente comando:

```
npx expo prebuild
```

3. Esto generará la estructura de Android e iOS necesaria para compilar tu aplicación.

Android

Descargar el SDK privado

a) Agregar el archivo `.aar` y el `setup.sdk.gradle`

1. Copia el archivo `.aar` del SDK y colócalo en la carpeta `android/app/libs/`. Si no existe, créala.
2. Copia el archivo `setup.sdk.gradle` en la ruta:

```
android/app/setup.sdk.gradle
```

3. En el archivo `android/app/build.gradle`, agrega:

```
apply from: 'setup.sdk.gradle'

dependencies {...
```

Esto configurará automáticamente la inclusión del `.aar` en el classpath del módulo `app`.

b) Verificar que el .aar está incluido como dependencia

El script `setup.sdk.gradle` se encargará de agregar el `.aar` como dependencia.

Si deseas hacerlo manualmente en lugar de usar el `setup.sdk.gradle`, asegurate de que en `android/app/build.gradle` tengas:

```
dependencies {

      implementation files('libs/waas-sdk-1.0.0.aar')

}
```

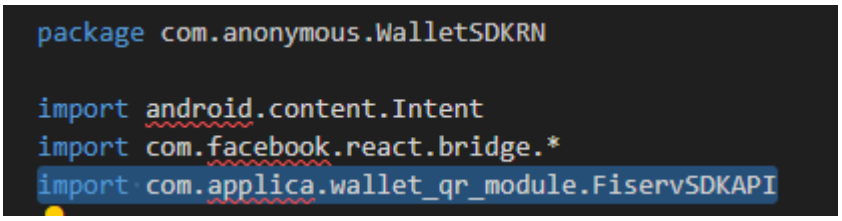
Usa este bloque **solo si no estás usando `setup.sdk.gradle`**.

c) Verificar la integración

1. Abre `MainApplication.java` o `MainActivity.java` en:

```
android/app/src/main/java/tu/paquete/principal/
```

2. Intenta importar alguna clase del SDK, por ejemplo:

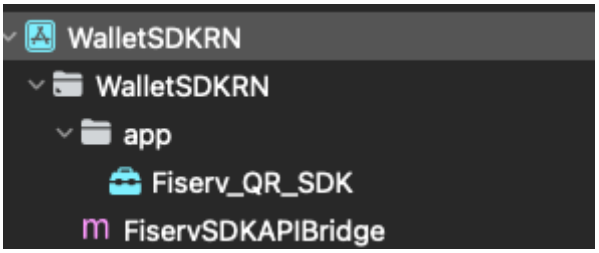


iOS

Descargar el SDK privado

a) Agregar el archivo `xcframework` al proyecto.

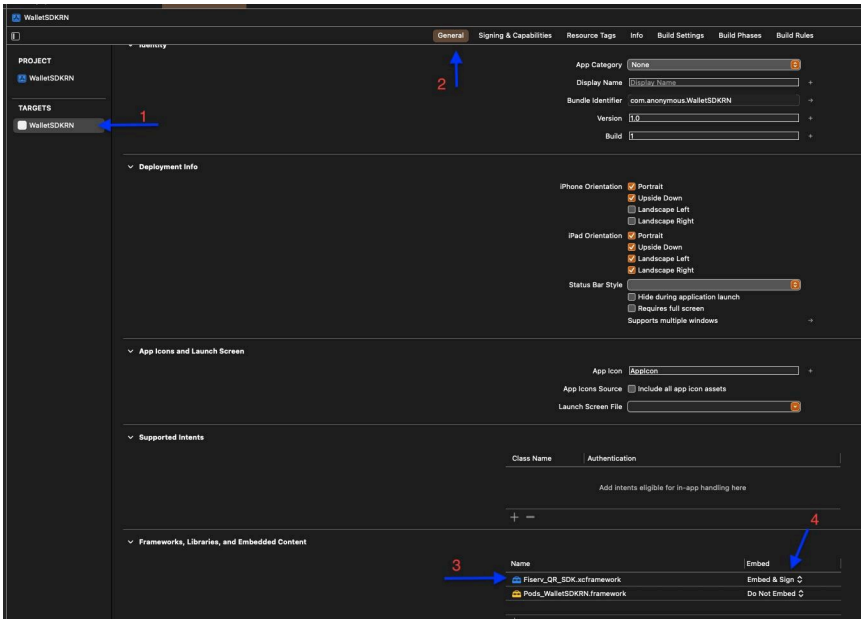
Incluye el archivo dentro del directorio correspondiente al build de iOS, preferentemente dentro de algún directorio independiente, para mantener un orden y facilitar la configuración de GIT (arrastra y suelta el archivo en el XCode para que lo embeda en el proyecto).




```
ios/app/Fiserv_QR_SDK.xcframework
```

b) Incluir la dependencia en el proyecto

En la configuración General del target, es necesario configurar el framework para que esté “embebido y firmado”.



c) Agregar los permisos necesarios para poder utilizar la cámara

En el archivo app.json, donde se encuentra la descripción del proyecto, es necesario incluir las siguientes líneas para requerir los permisos necesarios para utilizar la cámara en iOS:

```
"ios": {
    "infoPlist": {
      "NSCameraUsageDescription": "We need access to your camera"
    },
    "supportsTablet": true,
    "bundleIdentifier": "com.anonymous.WalletSDKRN",
    "deploymentTarget": "17"
  }
```

El texto resaltado incluye la petición de acceso a la cámara dentro del archivo “.plist” del build de iOS. El texto aquí debería ajustarse a la aplicación puntual.

Después de incluir éstas líneas, es necesario ejecutar el comando de ***prebuild*** en la raíz del proyecto, para que se incluyan los cambios de configuración en el proyecto de iOS.

Uso del Bridge en React Native

Para implementar un bridge para ambas plataformas, primero es necesario crear un archivo de JavaScript que funcionará dentro del proyecto de React. Éste archivo tendrá los métodos empleados por la aplicación, y será el que se deberá conectar luego, con los archivos “puente” de ambas plataformas.

A modo de ejemplo, a continuación se ve una implementación básica de dicho archivo de puente.

```
import { NativeModules } from 'react-native';

console.log("App is importing bridge module.");
console.log("NATIVE Modules:", NativeModules);

// Verifica que el bridge nativo esté vinculado correctamente
const { FiservSDKAPIBridge } = NativeModules;

if (!FiservSDKAPIBridge) {
  console.error("SDKAPIBridge no está disponible en NativeModules");
} else {
  console.log("SDKAPIBridge cargado correctamente:", FiservSDKAPIBridge);
}

if (!FiservSDKAPIBridge) {
  throw new Error(
    'FiservSDKAPIBridge is not available. Make sure the native module is linked correctly.'
  );
}
const FiservSDKAPI = {
  async initializeCheckout() {
    return FiservSDKAPIBridge.initializeCheckout();
  },
  /**
   * Configura la API con las credenciales necesarias.
   * @param {string} apiKey - Clave de la API.
   * @param {string} clientId - Valor del ClientId .
   * @param {string} duNumber - Valor del duNumber .
   * @param {string} duType - Valor del duType .
   * @param {string} walletId - Valor del WalletId.
   * @param {string} walletIdURLParam - Valor del walletIdURLParam.
   * @param {string} environment - Entorno: LOCAL, DEV, STAGE, PRODUCTION, etc.
   * @returns {Promise<string>} Promesa que se resuelve si la configuración es exitosa.
   */
  async setApiConfig(apiKey, duNumber, duType, clientId, walletId, walletIdURLParam, environment) {
    //async setApiConfig(apiKey, clientId, walletId, environment, host = null, port = null) {
    if (!['LOCAL', 'SIT', 'CAT', 'PRD'].includes(environment)) {
      throw new Error("Invalid environment. Use 'LOCAL', 'SIT', 'CAT', or 'PRD'.");
    }

    /*if (environment === 'LOCAL' && !host && !port) {
      throw new Error('Localhost must be provided when environment is LOCAL.');
    }*/
```

```
    return FiservSDKAPIBridge.setApiConfig(apiKey, duNumber, duType, clientId, walletId,
walletIdURLParam, environment);
  },
  /**
   * @param {string} paymentCheckoutModel - Modelo de datos json string para el checkout.
   * Inicia el flujo de checkout como una Activity con los datos proporcionados.
   * @returns {Promise<string>} Resultado del flujo de pago.
   */
  async startCheckoutActivity(paymentCheckoutModel) {
    if (typeof paymentCheckoutModel !== 'string') {
      throw new Error('paymentCheckoutModel must be a string.');
    }
    return FiservSDKAPIBridge.startCheckoutActivity(paymentCheckoutModel);
  },
  /**
  * @param {string} qrValue - String leido del qr.
  *
  * @returns {Promise<string>} Resultado del qr.
  */
  async decodeQRCodeJson(qrValue) {
    return FiservSDKAPIBridge.decodeQRCodeJson(qrValue);
  },

  /**
   *
   * @returns {Promise<string>} Objeto JSON en String con los medios de pagos del usuario
configurado.
   */
  async getPaymentMethods() {
    return FiservSDKAPIBridge.getPaymentMethods();
  },

  /**
   *
   * @param {string} qrId
   * @param {string} merchantRut
   * @param {string} qrScannedTime
   * @param {[string]} issuerCodeList
   *
   * @returns {Promise<[string]>} Promesa de respuesta con la lista de los issuerCodes aceptados por
el comercio.
   */
  async filterAllowedPaymentMethods(
    qrId,
    merchantRut,
    qrScannedTime,
    issuerCodeList
  ) {
    return FiservSDKAPIBridge.filterAllowedPaymentMethods(qrId, merchantRut, qrScannedTime,
issuerCodeList);
  },

  /**
   * Realiza un pago con un medio de pago procesado por Fiserv
   * @param {string} qrId
   * @param {string} qrScannedTime
   * @param {string} cardNumberM
   * @param {string} cardType
   * @param {string} issuerCode
   *
```

```
   * @returns {Promised<string>} Objeto JSON con el resultado de la operación
   */
  async paymentConfirmProcessed(
    qrId,
    qrScannedTime,
    cardNumberM,
    cardType,
    issuerCode
  ) {
    return FiservSDKAPIBridge.paymentConfirmProcessed(qrId, qrScannedTime, cardNumberM, cardType,
issuerCode);
  },

  /**
   * Realiza un pago con un medio de pago No procesado por Fiserv.
   * Se requieren todos los datos del medio de pago seleccionado.
   *
   * @param {string} qrId
   * @param {string} qrScannedTime
   * @param {string} cardHolder
   * @param {string} cardNumber
   * @param {string} cardType
   * @param {string} issuerCode
   * @param {string} dueDate
   * @param {string} cardCVV
   *
   * @returns {Promised<string>} Objeto JSON con el resultado de la operación
   */
  async paymentConfirm(
    qrId,
    qrScannedTime,
    cardHolder,
    cardNumber,
    cardType,
    issuerCode,
    dueDate,
    cardCVV
  ) {
    return FiservSDKAPIBridge.paymentConfirm(qrId, qrScannedTime, cardHolder, cardNumber, cardType,
issuerCode, dueDate, cardCVV);
  },

  /**
   *
   * @returns {Promised<string>} Devuelve un String con la hora actual, formateada para el SDK.
   */
  async getCurrentDateTime() {
    return FiservSDKAPIBridge.getCurrentDateTime();
  }
};

export default FiservSDKAPI;
```

En él se pueden ver los métodos disponibles en la integración de **FiservSDKAPI** para aplicaciones desarrolladas en React Native. Se proporciona una interfaz simplificada para la

comunicación con los SDK, incluyendo una validación mínima de parámetros antes de invocar el módulo de puente (bridge).

La aplicación utilizará el objeto **FiservSDKAPI** como intermediario para interactuar con los SDK, permitiendo una implementación transparente para los desarrolladores.

El método “initializeCheckout” es el encargado de inicializar el SDK, con todo el funcionamiento interno. Es importante llamar a éste método lo antes posible en el flujo de la aplicación, antes incluso, de configurar el SDK.

Luego, se ve el método “setApiConfig”. Éste método es el empleado para configurar el SDK para que se conecte con el servidor correctamente. En el ejemplo, se discriminan los parámetros necesarios, y el tipo de datos que necesitan. Para mayor independencia de plataforma, éstos parámetros se reducen a los modelos de datos estándar, como lo son “Strings”, dejando el parseo a modelos de datos particulares de cada plataforma dentro de los archivos del bridge de cada plataforma.

Decodifica un código QR mediante el método “decodeQRCodeJson”, extrayendo la información contenida y retornando los datos en formato JSON.
Es recomendable validar que el valor del QR sea correcto antes de enviarlo al SDK para evitar errores en el procesamiento.

Obtiene la lista de medios de pagos configurados para el usuario en el sistema de Fiserv, mediante el método “getPaymentMethods” .
El resultado es una cadena en formato JSON con los métodos de pago disponibles, útil para mostrarlos en la interfaz de usuario.

Filtra los medios de pago permitidos por el comercio en función del código QR escaneado y otros datos relevantes con la función “filterAllowedPaymentMethods”.
Se recomienda validar que la lista de emisores (`issuerCodeList`) no esté vacía antes de realizar la consulta, evitando solicitudes innecesarias al SDK.

Procesa un pago con un medio de pago validado y procesado directamente por Fiserv a través del “paymentConfirmProcessed” con el documentNumber proporcionado anteriormente.
Es imprescindible validar todos los parámetros requeridos antes de enviarlos al SDK.

Procesa un pago con un medio de pago no procesado directamente por Fiserv mediante el “paymentConfirm”.
Se recomienda validar los datos de la tarjeta y verificar la fecha de vencimiento antes de realizar la solicitud.

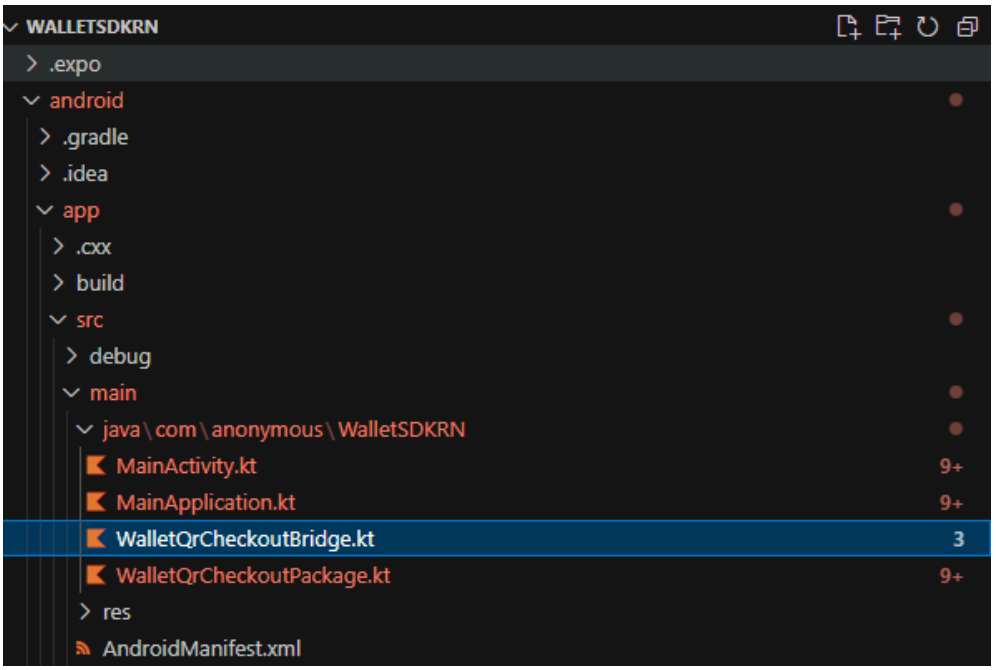
También se ve expuesta la función “getCurrentDateTime” que permite obtener la fecha y hora actual en un formato compatible con el SDK.
Este método es útil para luego utilizar correctamente las demás funciones que requieren este valor.

Finalmente, se ve un método “startCheckoutActivity”, encargado de inicializar el flujo de pagos. Éste método también recibe como parámetro un valor del tipo String, el cual debe contener los medios de pagos descritos en la implementación nativa de los SDK. Éste parámetro debe ser una representación del JSON que contenga toda la información necesaria, convertido a un único String. Internamente, los archivos del bridge de cada plataforma se encargan de convertir éstos string a los modelos de datos necesarios.

Implementación del Bridge nativo en Android

Creación del Bridge

1. Crea un archivo llamado `WalletQrCheckoutBridge.kt` en el paquete principal de tu aplicación.



2. Copia y pega el contenido del archivo proporcionado para `WalletQrCheckoutBridge`.

```
import android.content.Intent
import com.facebook.react.bridge.*
import androidx.compose.ui.graphics.Color
import com.anonymous.WalletSDKRN.R
import com.applica.wallet_qr_module.FiservSDKAPI
import com.applica.wallet_qr_module.configs.Environment
import com.applica.wallet_qr_module.configs.ServiceConfig
import com.applica.wallet_qr_module.configs.CheckoutServiceConfig
import com.applica.wallet_qr_module.domain.services.PaymentServiceListener
import com.applica.wallet_qr_module.domain.services.PaymentServiceManager
import com.applica.wallet_qr_module.configs.ResultPaymentListener
import com.applica.wallet_qr_module.configs.Approval
import com.applica.wallet_qr_module.configs.error.ErrorInfo

class FiservSDKAPIBridge(reactContext: ReactApplicationContext) :
ReactContextBaseJavaModule(reactContext) {
    private val appContext: ReactApplicationContext = reactContext

    @ReactMethod
```

```
    fun initializeCheckout() {
        FiservSDKAPI.initialize(appContext)
        /*FiservSDKAPI.setConfig(
            moduleColors = ModuleColors(
                primary = Color(0xFF008000)
            ),
            moduleDrawableResources = ModuleDrawableResources(logo =
R.drawable.ic_logo_demo),
            moduleStringResources = ModuleStringResources(
                scannerQrPlaceholderScreen = ScannerQrScreenStringResources(title =
R.string.scanner_title)
            )
        )*/
    }

    @ReactMethod
    fun setApiConfig(apiKey: String,documentNumber: String, documentType: String,
clientId: String, walletId: String,walletIdURLParam:String, environment: String,
promise: Promise) {
        try {
            // Configuración del environment con soporte para LOCAL
            val env =  Environment.valueOf(environment)

            val serviceConfig = ServiceConfig(apiKey = apiKey, clientId = clientId,
documentNumber = documentNumber, documentType = documentType, walletId = walletId,
walletIdURLParam = walletIdURLParam, environment = env)
            val checkoutServiceConfig = CheckoutServiceConfig(serviceConfig)
            FiservSDKAPI.setApiConfig(checkoutServiceConfig)

            promise.resolve("Configuration set successfully")
        } catch (e: Exception) {
            promise.reject("CONFIG_ERROR", "Failed to set API config", e)
        }
    }

    @ReactMethod
    fun startCheckoutActivity(paymentCheckoutJsonString: String, promise: Promise)
{
        try {

            // Verificar que currentActivity no sea nulo
            val activity = currentActivity
            if (activity == null) {
                promise.reject("ACTIVITY_ERROR", "Current activity is null")
                return
            }

            // Configurar el listener para los resultados del pago
            val resultPaymentListener = object : ResultPaymentListener {
                override fun onApproval(approval: Approval) {
```

```
                    val status = approval.paymentResult.state
                    val amount = approval.paymentResult.amountDetails?.amount
                    promise.resolve("Payment $status: $amount")
                }

                override fun onCancel() {
                    promise.resolve("Payment cancelled")
                }

                override fun onError(errorInfo: ErrorInfo) {
                    promise.reject("PAYMENT_ERROR", "Payment error occurred:
${errorInfo.reason}")
                }
            }

            // Iniciar el flujo de pago como Activity
            FiservSDKAPI.startCheckout(
                context = activity.applicationContext,
                paymentCheckoutJsonString = paymentCheckoutJsonString,
                resultPaymentListener = resultPaymentListener
            )

        } catch (e: Exception) {
            promise.reject("CHECKOUT_ERROR", "Failed to start checkout activity:
${e.message.toString()}", e)
        }
    }

    @ReactMethod
    fun paymentConfirm(
        qrId: String,
        qrScannedTime:String,
        cardHolder: String,
        cardNumber: String,
        cardType: String,
        issuerCode: String,
        dueDate: String,
        cardCvv: String,
        promise: Promise
    ) {
        try {
            FiservSDKAPI.getPaymentServiceManager().paymentConfirm(
                qrId = qrId,
                cardNumber = cardNumber,
                dueDate = dueDate,
                cardCvv = cardCvv,
                cardHolder = cardHolder,
                cardType = cardType,
                issuerCode = issuerCode,
                qrScannedTime = qrScannedTime,
```

```
                listener = object : PaymentServiceListener<String> {
                    override fun onSuccess(response: String) {
                        promise.resolve("Pago aprobado: $response")
                    }

                    override fun onFailure(error: String) {
                        promise.reject("CHECKOUT_ERROR", "Failed to call payment
confirm service: $error")
                    }
                })
        }catch (e: Exception) {
            promise.reject("CHECKOUT_ERROR", "Failed to call payment confirm
service: ${e.message.toString()}", e)
        }
    }

    @ReactMethod
    fun paymentConfirmProcessed(
        qrId: String,
        qrScannedTime: String,
        cardNumberM: String,
        cardType: String,
        issuerCode: String,
        promise: Promise
    ){
        try{
            FiservSDKAPI.getPaymentServiceManager().paymentConfirm(
                qrId = qrId,
                cardNumberM = cardNumberM,
                cardType = cardType,
                issuerCode = issuerCode,
                qrScannedTime = qrScannedTime,
                listener = object : PaymentServiceListener<String> {
                    override fun onSuccess(response: String) {
                        promise.resolve("Pago aprobado: $response")
                    }

                    override fun onFailure(error: String) {
                        promise.reject("CHECKOUT_ERROR", "Failed to call payment
confirm service: $error")
                    }
                })
        }catch (e: Exception) {
            promise.reject("CHECKOUT_ERROR", "Failed to call payment confirm
service: ${e.message.toString()}", e)
        }
    }

    @ReactMethod
    fun getPaymentMethods(promise: Promise) {
```

```
        try {
            FiservSDKAPI.getPaymentServiceManager().getPaymentMethods(listener =
object :
                PaymentServiceListener<String> {
                override fun onSuccess(response: String) {
                    promise.resolve(response)
                }

                override fun onFailure(error: String) {
                    promise.reject("CHECKOUT_ERROR", "Failed to call Payment
methods service: $error")
                }
            })
        }catch (e: Exception) {
            promise.reject("CHECKOUT_ERROR", "Failed to call Payment methods
service ${e.message.toString()}", e)
        }
    }

    @ReactMethod
    fun filterAllowedPaymentMethods(
        qrId: String,
        merchantRut: String,
        qrScannedTime: String,
        issuerCodeList: ReadableArray,
        promise: Promise
    ) {
        try {
            val issuerList = mutableListOf<String>()
            for (i in 0 until issuerCodeList.size()) {
                issuerList.add(issuerCodeList.getString(i) ?: "")
            }
            FiservSDKAPI.getPaymentServiceManager().filterAllowedPaymentMethods(
                qrId = qrId,
                merchantRut = merchantRut,
                qrScannedTime = qrScannedTime,
                issuerCodeList = issuerList,
                listener = object : PaymentServiceListener<String> {
                    override fun onSuccess(response: String) {
                        promise.resolve(response)
                    }

                    override fun onFailure(error: String) {
                        promise.reject("CHECKOUT_ERROR", "Failed to call Payment
methods accepted by merchant service: $error")
                    }
                }
            )
        }catch (e: Exception) {
```

```
            promise.reject("CHECKOUT_ERROR", "Failed to call Payment methods
accepted by merchant service ${e.message.toString()}", e)
        }
    }

    @ReactMethod
    fun getCurrentDateTime(promise: Promise) {
        val currentDateTime = FiservSDKAPI.getCurrentDateTime()
        promise.resolve(currentDateTime)
    }

    @ReactMethod
    fun decodeQRCodeJson(qrCode: String, promise: Promise) {
        val qrResult = FiservSDKAPI.decodeQRCodeJson(qrCode = qrCode)
        promise.resolve(qrResult)
    }

    override fun getName(): String {
        return "FiservSDKAPIBridge"
    }
}
```

1. initializeCheckout

Descripción:
Este método inicializa el SDK nativo y debe ser llamado antes de interactuar con cualquier funcionalidad del flujo de pago o configuración. Además de preparar el SDK para su uso, permite personalizar diversos aspectos visuales y de contenido. Por ejemplo, es posible configurar los colores del módulo mediante la clase `ModuleColors`, como el color primario (`primary`) definido en este caso con el valor `Color(0xFF008000)`. También se pueden establecer recursos gráficos personalizados usando `ModuleDrawableResources`, como un logotipo (`logo`) asignado a `R.drawable.ic_logo_demo`. Asimismo, los textos de las pantallas pueden ser ajustados utilizando `ModuleStringResources`, como en el caso del título del escáner QR, configurado a través de `ScannerQrScreenStringResources` con el recurso `R.string.scanner_title`. Estas opciones permiten adaptar la apariencia y los textos del SDK a las necesidades específicas del proyecto.

Uso típico:

- Se utiliza una vez al inicio del ciclo de vida de la aplicación, preferiblemente en el método `useEffect` del componente raíz.

2. setApiConfig

Descripción:
Este método configura el SDK con las credenciales necesarias y define el entorno operativo, como `LOCAL`, `SIT`, `CAT` o `PRD`. Es fundamental para garantizar que las solicitudes se autentiquen correctamente y que la aplicación se conecte al entorno adecuado. Además, ofrece soporte para configuraciones personalizadas en el entorno `LOCAL`, como el host y el puerto del servidor.

Parámetros:

- `apiKey`: Clave única para autenticar las solicitudes en el sistema.
- `clientId`: Identificador del cliente que realiza la configuración.
- `walletId`: Identificador único de la billetera del cliente.
- `walletIdURLParam`: Identificador único de la billetera del cliente.
- `documentNumber`: Identificador único del usuario de la billetera.
- `documentType`: Tipo de documento del usuario de la billetera (`DU`).
- `environment`: Entorno donde se ejecutará la aplicación (`LOCAL`, `DEV`, etc.).
- `host` (opcional): Dirección del servidor (solo para el entorno `LOCAL`).
- `port` (opcional): Puerto del servidor (solo para el entorno `LOCAL`).

Validaciones:

- Si el entorno es `LOCAL`, se deben proporcionar `host` y `port`.

3. startCheckoutActivity

Descripción:
Este método lanza el flujo de pago utilizando el SDK nativo como una `Activity`. Recibe los datos de pago en formato JSON, los procesa, y maneja los eventos resultantes como éxito, cancelación o error.

Parámetros:

- `paymentCheckoutJsonString`: Cadena JSON que contiene los datos necesarios para el proceso de pago. Incluye detalles como métodos de pago disponibles, información del titular de la wallet, entre otros.
- `promise`: Promesa para notificar el resultado del flujo (éxito, cancelación o error).

Flujo Interno:

1. Obtiene la actividad actual (`currentActivity`).
2. Configura un listener (`ResultPaymentListener`) para manejar los resultados:
    - **`onApproval`**: Se invoca cuando el pago se aprueba exitosamente.
    - **`onCancel`**: Se invoca si el usuario cancela el proceso.
    - **`onError`**: Se invoca si ocurre un error.
3. Inicia el flujo de pago con `WalletQrCheckout.startCheckout`.

4. paymentConfirm

Descripción:
Confirma un pago utilizando una tarjeta no procesada por Fiserv.

Parámetros:

- `qrId` (String): Identificador del código QR.
- `qrScannedTime` (String): Hora en que se escaneó el código QR.
- `cardHolder` (String): Nombre del titular de la tarjeta.
- `cardNumber` (String): Número de la tarjeta.
- `cardType` (String): Tipo de tarjeta (ej. VISA, MasterCard).
- `issuerCode` (String): Código del emisor de la tarjeta.
- `dueDate` (String): Fecha de vencimiento de la tarjeta.
- `cardCvv` (String): Código de seguridad de la tarjeta.
- `promise` (Promise): Objeto de promesa de React Native para manejar la respuesta.

Validaciones:

- Si la llamada al servicio falla, se rechaza la promesa con el código `CHECKOUT_ERROR`.

5. paymentConfirmProcessed

Descripción:
Confirma un pago utilizando una tarjeta procesada por Fiserv.

Parámetros:

- `qrId` (String): Identificador del código QR.
- `qrScannedTime` (String): Hora en que se escaneó el código QR.
- `cardHolder` (String): Nombre del titular de la tarjeta.
- `cardNumberM` (String): Número de la tarjeta procesada.
- `cardType` (String): Tipo de tarjeta (ej. VISA, MasterCard).
- `issuerCode` (String): Código del emisor de la tarjeta.
- `promise` (Promise): Objeto de promesa de React Native para manejar la respuesta.

Validaciones:

- Si la llamada al servicio falla, se rechaza la promesa con el código `CHECKOUT_ERROR`.

6. getPaymentMethods

Descripción:
Obtiene la lista de métodos de pago disponibles procesados por Fiserv. para el documentNumber proporcionado.

Parámetros:

- `promise` (Promise): Objeto de promesa de React Native para manejar la respuesta.

Validaciones:

- Si la llamada al servicio falla, se rechaza la promesa con el código `CHECKOUT_ERROR`.

6. filterAllowedPaymentMethods

Descripción:
Filtra los métodos de pago permitidos por el comercio que generó el QR.

Parámetros:

- `qrId` (String): Identificador del código QR.
- `merchantRut` (String): RUT del comercio.
- `qrScannedTime` (String): Hora en que se escaneó el código QR.
- `issuerCodeList` (ReadableArray): Lista de códigos de emisores de tarjetas permitidos.
- `promise` (Promise): Objeto de promesa de React Native para manejar la respuesta.

Validaciones:

- Se valida que `issuerCodeList` sea un array válido antes de procesarlo.
- Si la llamada al servicio falla, se rechaza la promesa con el código `CHECKOUT_ERROR`.

7. decodeQRCodeJson

Descripción:
Decodifica un código QR en formato JSON utilizando el SDK de Fiserv.

Parámetros:

- `qrCode` (String): Código QR a decodificar.

- `promise` (Promise): Objeto de promesa de React Native para manejar la respuesta.

8. getCurrentDateTime

Descripción:
Obtiene la fecha y hora actual del sistema desde el SDK de Fiserv y lo formatea en base al tipo esperado dentro del SDK.

Parámetros:

- `promise` (Promise): Objeto de promesa de React Native para manejar la respuesta.

Creación del Paquete del Bridge

1. Crea un archivo llamado `WalletQrCheckoutPackage.kt` en el mismo paquete que el bridge.
2. Copia y pega el contenido del archivo proporcionado para `WalletQrCheckoutPackage`.

```
import com.facebook.react.ReactPackage
import com.facebook.react.bridge.NativeModule
import com.facebook.react.bridge.ReactApplicationContext
import com.facebook.react.uimanager.ViewManager

class WalletQrCheckoutPackage : ReactPackage {
    override fun createNativeModules(reactContext:
ReactApplicationContext): List<NativeModule> {
        return listOf(WalletQrCheckoutBridge(reactContext))
    }

    override fun createViewManagers(reactContext:
ReactApplicationContext): List<ViewManager<*, *>> {
        return emptyList()
    }
}
```

Descripción:

El `WalletQrCheckoutPackage` es una clase que actúa como el punto de entrada para registrar los módulos nativos en React Native. Este archivo es esencial para que el bridge (`WalletQrCheckoutBridge`) sea reconocido y accesible desde el código JavaScript.

Configuración de MainApplication

1. Abre el archivo `MainApplication.kt` ubicado en `android/app/src/main/java/<paquete_de_tu_app>/`.
2. Asegúrate de que el paquete del Bridge esté importado correctamente:

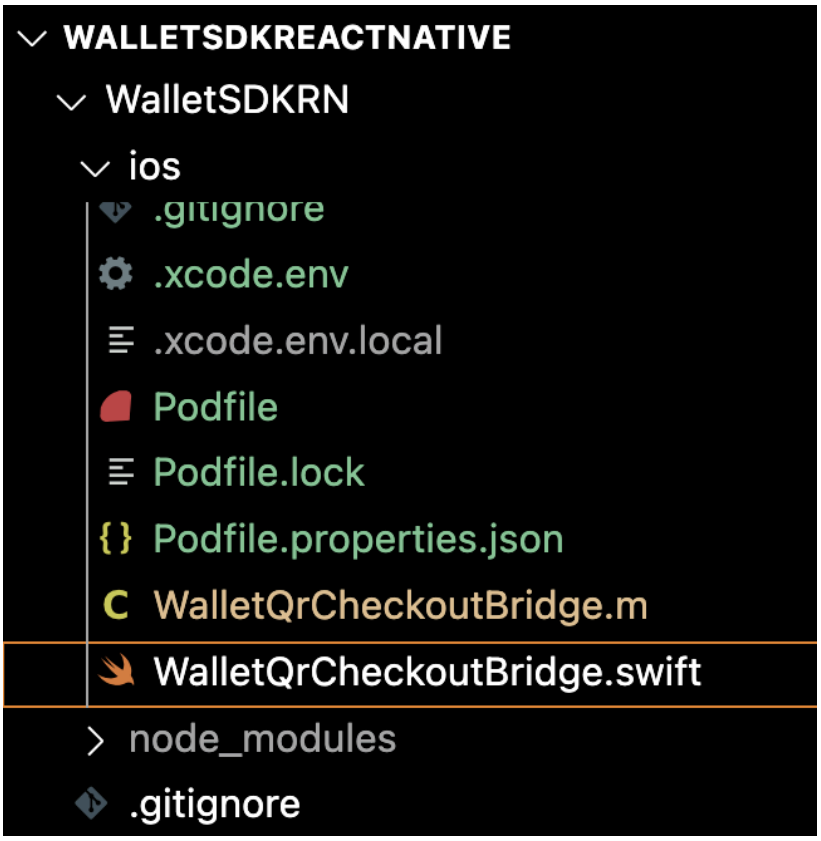
```
import com.anonymous.WalletSDKRN.WalletQrCheckoutPackage
```

3. Agrega el paquete del bridge al método `getPackages`:

```
 override val reactNativeHost: ReactNativeHost =
ReactNativeHostWrapper(
        this,
        object : DefaultReactNativeHost(this) {
            override fun getPackages(): List<ReactPackage> {
                // Obtiene los paquetes predeterminados
                val packages =
PackageList(this).packages.toMutableList()

                // Agrega el WalletQrCheckoutPackage al listado de
paquetes manualmente
                packages.add(WalletQrCheckoutPackage())
                return packages
            }

            override fun getJSMainModuleName(): String =
".expo/.virtual-metro-entry"

            override fun getUseDeveloperSupport(): Boolean =
BuildConfig.DEBUG

            override val isNewArchEnabled: Boolean =
BuildConfig.IS_NEW_ARCHITECTURE_ENABLED
            override val isHermesEnabled: Boolean =
BuildConfig.IS_HERMES_ENABLED
        }
    )
```

Implementación del Bridge nativo en iOS

Para que el puente funcione en el build de iOS, es necesario crear el archivo en Swift y un archivo “.m” que harán de puente nativo.



Configuración del archivo “.swift”.

La clase, en el ejemplo, se llama “FiservSDKAPIBridge”, y debe extenderse con los protocolos de NSObject y RTCBridgeModule.

Luego, es necesario agregar el método estático “moduleName”, el cual devuelve el nombre del bridge. Esto es importante para que el Bridge sepa cuál es el archivo que corresponde al bridge.

```
import Foundation
import React
//import FISERV_SDK_API    //-> Import del SDK. En éste ejemplo están los dummies incluidos en el proyecto, y no hace falta.

@objc(FiservSDKAPIBridge)
class FiservSDKAPIBridge: NSObject, RCTBridgeModule {

  static func moduleName() -> String {
      return "FiservSDKAPIBridge"
```

```
}

@objc
func initializeCheckout(
  _ resolve: @escaping RCTPromiseResolveBlock,
  reject: @escaping RCTPromiseRejectBlock
) {
  resolve("Initialize successful")
}

@objc
func setApiConfig(
  _ apiKey: String,
  duNumber: String,
  duType: String,
  clientId: String,
  walletId: String,
  walletIdURLParam: String,
  environment: String,
  resolve: @escaping RCTPromiseResolveBlock,
  reject: @escaping RCTPromiseRejectBlock
) {
  print("***** API Config for \(environment)")
  // Validación de entorno
  guard ["LOCAL", "CAT", "SIT", "PRD"].contains(environment) else {
    reject("INVALID_ENV", "Invalid environment", nil)
    return
  }

  let env: EnvironmentFlavor
  switch environment {
  case "LOCAL":
    env = .local
  case "CAT":
    env = .cat
  case "SIT":
    env = .sit
  case "PRD":
    env = .production
  default:
    reject("INVALID_ENV", "Invalid environment", nil)
    return
  }
  // Configura la API usando el SDK
  do {
      let config = SDKMinimumConfiguration(
        environment: env,
        apiKey: apiKey,
        duType: duType,
        duNumber: duNumber,
        walletId: walletId,
        walletIdURLParam: walletIdURLParam,
        clientId: clientId
      )
```

```
      FiservSDKAPI.configSDK(config)
      FiservSDKAPI.startServiceManager()
      resolve("SDK Configurated")
    } catch {
      reject("CONFIG_ERROR", "Failed to configure API", error)
    }
  }


  @objc
  func startCheckoutActivity(
    _ paymentCheckoutModel: String,
    resolve: @escaping RCTPromiseResolveBlock,
    reject: @escaping RCTPromiseRejectBlock
  ) {
    DispatchQueue.main.async {
      guard
          let windowScene = UIApplication.shared.connectedScenes.first as? UIWindowScene,
          let keyWindow = windowScene.windows.first(where: { $0.isKeyWindow }),
          let rootViewController = keyWindow.rootViewController
      else {
          reject("no_root_vc", "No se encontró RootViewController", nil)
          return
      }

      let paymentResultDelegate = PaymentResultDelegateWrapper(
          onApproval: { approvalResult in
              print("Payment approved")
              resolve(approvalResult)
          },
          onCancel: {
              print("Payment cancelled")
              reject("payment_cancelled", "El usuario canceló el flujo de pago", nil)
          },
          onError: { errorInfo in
              print("Payment error")
            reject("payment_error", errorInfo.message, nil)
          }
      )

      guard let vc = FiservSDKAPI.getPaymentFlowUIViewControllerFromJson(paymentCheckoutModel:
paymentCheckoutModel, paymentResultDelegate: paymentResultDelegate)
      else {
        reject("CONFIG_ERROR", "Something failed while loading the SDK", nil)
        return
      }

      vc.modalPresentationStyle = .formSheet

      rootViewController.present(vc, animated: true)
    }
  }
```

```
  @objc
  func decodeQRCodeJson(
    _ qrCode: String,
    resolve: @escaping RCTPromiseResolveBlock,
    reject: @escaping RCTPromiseRejectBlock
  ) {
    let qrResult: String = FiservSDKAPI.decodeQRCodeJson(qrCode) ?? ""
      resolve(qrResult)
  }

  @objc
  func getPaymentMethods(
    _ resolve: @escaping RCTPromiseResolveBlock,
    reject: @escaping RCTPromiseRejectBlock
  ) {
    Task {
      do {
        let paymentMethods: String = try await FiservSDKAPI.getPaymentMethods()
        resolve(paymentMethods)
      } catch {
        reject("GENERAL ERROR", error.localizedDescription, error)
      }
    }
  }

  @objc
  func filterAllowedPaymentMethods(
    _ qrId: String,
    merchantRut: String,
    qrScannedTime: String,
    issuerCodeList: [String],
    resolve: @escaping RCTPromiseResolveBlock,
    reject: @escaping RCTPromiseRejectBlock
  ) {
    Task {
      do {
        let allowedPaymentMethods: [String] = try await
FiservSDKAPI.filterAllowedPaymentMethods(for: qrId, with: merchantRut, qrScannedTime: qrScannedTime,
issuerCodeList: issuerCodeList)
        resolve(allowedPaymentMethods)
      } catch {
        reject("GENERAL ERROR", error.localizedDescription, error)
      }
    }
  }

  @objc
  func paymentConfirmProcessed(
    _ qrId: String,
    qrScannedTime: String,
    cardNumberM: String,
    cardType: String,
    issuerCode: String,
    resolve: @escaping RCTPromiseResolveBlock,
    reject: @escaping RCTPromiseRejectBlock
  ) {
    Task {
      do {
```

```
        let paymentResponse: String = try await FiservSDKAPI.paymentConfirm(for: qrId,
qrScannedTime: qrScannedTime, cardHolder: "", cardNumberM: cardNumberM, cardType: cardType,
issuerCode: issuerCode)
        resolve(paymentResponse)
      } catch {
        reject("GENERAL ERROR", error.localizedDescription, error)
      }
    }
  }


  @objc
  func paymentConfirm(
    _ qrId: String,
    qrScannedTime: String,
    cardHolder: String,
    cardNumber: String,
    cardType: String,
    issuerCode: String,
    dueDate: String,
    cardCVV: String,
    resolve: @escaping RCTPromiseResolveBlock,
    reject: @escaping RCTPromiseRejectBlock
  ) {
    Task {
      do {
        let paymentResponse: String = try await FiservSDKAPI.paymentConfirm(for: qrId,
qrScannedTime: qrScannedTime, cardHolder: cardHolder, cardNumber: cardNumber, cardType: cardType,
issuerCode: issuerCode, dueDate: dueDate, cardCVV: cardCVV)
        resolve(paymentResponse)
      } catch {
        reject("GENERAL ERROR", error.localizedDescription, error)
      }
    }
  }

  @objc
  func getCurrentDateTime(
    _ resolve: @escaping RCTPromiseResolveBlock,
    reject: @escaping RCTPromiseRejectBlock
  ) {
    resolve(FiservSDKAPI.getCurrentDateTime())
  }
}


class PaymentResultDelegateWrapper: NSObject, PaymentResultDelegate {
   let onApprovalCallback: (Fiserv_QR_SDK.Approval) -> Void
   let onCancelCallback: () -> Void
   let onErrorCallback: (Fiserv_QR_SDK.ErrorInfo) -> Void

   init(onApproval: @escaping (Fiserv_QR_SDK.Approval) -> Void,
        onCancel: @escaping () -> Void,
        onError: @escaping (Fiserv_QR_SDK.ErrorInfo) -> Void) {
       self.onApprovalCallback = onApproval
       self.onCancelCallback = onCancel
       self.onErrorCallback = onError
   }
```

```
    func onApproval(_ approvalResult: Fiserv_QR_SDK.Approval) {
        onApprovalCallback(approvalResult)
    }

    func onCancel() {
        onCancelCallback()
    }

    func onError(_ errorInfo: Fiserv_QR_SDK.ErrorInfo) {
        onErrorCallback(errorInfo)
    }
}
```

Como se ve en la implementación del bridge, aquí es donde se importa la librería del SDK, y donde se utiliza código nativo de Swift para generar la interfaz con el SDK de pagos.

Las funciones coinciden con nombre y parámetros con respecto a las formuladas en el archivo de puente de JavaScript, empleado por la aplicación de ReactNative.

Es importante anotar cada método con “@objc”, para que sean compatibles con objective-c. Como se puede observar en el bloque de código, aquí hay una mínima análisis de datos y adaptación de los mismos para configurar el SDK.
Otro dato de suma importancia, es que el primer parámetro que recibe cada método, vaya sin label, como se muestra en el código, utilizando el símbolo “_” antes del nombre del parámetro.

En el caso del método de “startCheckoutActivity”, el propio método del SDK es el encargado de parsear el string que contiene la información de los métodos de pagos, y convertirlo a un modelo de datos nativo, pero es posible hacerlo aquí, si se desea hacer alguna validación extra.

El método de pago empleado devuelve un ViewController, el cual es ajustado y presentado dentro del método del bridge. Aquí se puede dimensionar o agregar alguna decoración extra.

Configuración del archivo “.m”.

En éste archivo, es necesario referenciar los métodos declarados en el archivo de Swift, para que sean accedidos por métodos de objective-c.

Se puede observar la definición del módulo “externo” FiservSDKAPIBridge, que hace referencia a nuestro archivo .swift, y la declaración de cada método incluído en el mismo, junto con la lista de parámetros que necesitan.

```
//
//  WalletQrCheckoutBridge.m
//  WalletSDKRN
```

```
//
//  Created by Ariel Scarafia on 26/12/2024.
//

#import <Foundation/Foundation.h>

#import <React/RCTBridgeModule.h>

NS_ASSUME_NONNULL_BEGIN

@interface
RCT_EXTERN_MODULE(FiservSDKAPIBridge, NSObject)

RCT_EXTERN_METHOD(
                  initializeCheckout:
                  (RCTPromiseResolveBlock)resolve
                  reject:(RCTPromiseRejectBlock)reject
                  )

RCT_EXTERN_METHOD(
                  setApiConfig:
                  (NSString *)apiKey
                  duNumber:(NSString *)duNumber
                  duType:(NSString *)duType
                  clientId:(NSString *)clientId
                  walletId:(NSString *)walletId
                  walletIdURLParam:(NSString *)walletIdURLParam
                  environment:(NSString *)environment
                  resolve:(RCTPromiseResolveBlock)resolve
                  reject:(RCTPromiseRejectBlock)reject
                  )

RCT_EXTERN_METHOD(
                  startCheckoutActivity:
                  (NSString *)paymentCheckoutModel
                  resolve:(RCTPromiseResolveBlock)resolve
                  reject:(RCTPromiseRejectBlock)reject
                  )

RCT_EXTERN_METHOD(
                  decodeQRCodeJson:
                  (NSString *)qrCode
                  resolve:(RCTPromiseResolveBlock)resolve
                  reject:(RCTPromiseRejectBlock)reject
                  )

RCT_EXTERN_METHOD(
                  getPaymentMethods:
                  (RCTPromiseResolveBlock)resolve
                  reject:(RCTPromiseRejectBlock)reject
                  )

RCT_EXTERN_METHOD(
                  filterAllowedPaymentMethods:
```

```
                  (NSString *)qrId
                  merchantRut:(NSString *)merchantRut
                  qrScannedTime:(NSString *)qrScannedTime
                  issuerCodeList:(NSArray<NSString *> *)issuerCodeList
                  resolve:(RCTPromiseResolveBlock)resolve
                  reject:(RCTPromiseRejectBlock)reject
                  )

RCT_EXTERN_METHOD(
                  paymentConfirmProcessed:
                  (NSString *)qrId
                  qrScannedTime:(NSString *)qrScannedTime
                  cardNumberM:(NSString *)cardNumberM
                  cardType:(NSString *)cardType
                  issuerCode:(NSString *)issuerCode
                  resolve:(RCTPromiseResolveBlock)resolve
                  reject:(RCTPromiseRejectBlock)reject
                  )

RCT_EXTERN_METHOD(
                  paymentConfirm:
                  (NSString *)qrId
                  qrScannedTime:(NSString *)qrScannedTime
                  cardHolder:(NSString *)cardHolder
                  cardNumber:(NSString *)cardNumber
                  cardType:(NSString *)cardType
                  issuerCode:(NSString *)issuerCode
                  dueDate:(NSString *)dueDate
                  cardCVV:(NSString *)cardCVV
                  resolve:(RCTPromiseResolveBlock)resolve
                  reject:(RCTPromiseRejectBlock)reject
                  )

RCT_EXTERN_METHOD(
                 getCurrentDateTime:
                 (RCTPromiseResolveBlock)resolve
                 reject:(RCTPromiseRejectBlock)reject
                 )

@end

NS_ASSUME_NONNULL_END
```

Aquí se debe declarar el módulo con RCT_EXTERN_MODULE, pasándole como parámetros el nombre del archivo del bridge, junto con un NSObject.

Además, es necesario definir cada método con RCT_EXTERN_METHOD. Nótese que aquí, para reflejar la NO inclusión del label del primer parámetro, se omite también, el label del primer parámetro de cada función.

Inclusión del módulo del bridge.

Finalmente, es necesario revisar el archivo generado por React-Native del header del puente. Éste archivo se encuentra dentro de la carpeta del proyecto, y su nombre comienza con el mismo nombre de proyecto, continuado con “-Bridging-Header.h”. En nuestro caso, dicho archivo es:

```
ios/WalletSDKRN/WalletSDKRN-Briding-Header.h
```

Dentro de éste archivo, debería estar un import del módulo de bridge de React, como se muestra a continuación:

```
//
//  Use this file to import your target's public headers that you would like to expose to Swift.
//
#import <React/RCTBridgeModule.h>
```

Ejemplo de aplicación empleando el Bridge, en React Native

Finalmente, se incluye un ejemplo de una aplicación muy básica, incluyendo el módulo del bridge, configurándolo, y disparando el flujo de pago con un botón.

```
import React, { useEffect } from 'react';
import { Alert, Button, StyleSheet, Text, View } from 'react-native';
import FiservSDKAPI from './FiservSDKAPI';
import { useState } from 'react';

import EMVCoQRData from "./EMVCoQR";
import parseProcessedPaymentMethods from "./JsonParser"

export default function App() {

  let [paymentMethods, setPaymentMethods] = useState([]);
  let [qrData, setQRData] = useState(null);
  let [qrScannedTime, setQRScannedTime] = useState([]);

  useEffect(() => {
    console.log("Métodos de pago actualizados:", paymentMethods);
  }, [paymentMethods]);

  useEffect(() => {
    console.log("setQRData:", qrData);
  }, [qrData]);

  useEffect(() => {
    // Configura la API al iniciar la aplicación
    async function configureApi() {
      try {
        console.log("App Starts here ********")
        await FiservSDKAPI.initializeCheckout()
          .then(response => {
            console.log(response); // Debería imprimir "test successful"
          }); // Aquí se instancia el SDK si es necesario

        // Cambia 'YOUR_API_KEY' y '123456789' por tus valores reales
        await FiservSDKAPI.setApiConfig(
          'validApiKey123',
          '12345678',
          'DU',
          'validClientId123',
          'walletId',
          'LOCAL',
          '192.168.1.8',
          '8080');
        console.log('API configurada correctamente en entorno LOCAL');
      } catch (error) {
        console.error('Error al configurar la API:', error.message);
      }
    }
    configureApi();
    getPaymentMethods();
  }, []);

  const handleCheckout = async () => {
```

```
    const paymentModel = {
      availablePaymentMethods: [
        {
          cardNumber: '5494000000004379',
          expDate: '12/25',
          cardHolder: 'John Doe',
          brandCode: 1,
          issuerCode: "75",
          isSelected: true,
        },
        {
          cardNumber: '5494000000009773',
          expDate: '12/25',
          cardHolder: 'John Doe',
          cvv: 123,
          brandCode: 2,
          issuerCode: "80",
          isSelected: true,
        },
      ]
    };

    try {
      // Inicia el flujo de pago como Activity
      const paymentModelString = JSON.stringify(paymentModel);
      console.log(paymentModelString)
      const result = await FiservSDKAPI.startCheckoutActivity(paymentModelString);
      console.log(result)
      Alert.alert('Resultado del Pago', result);
    } catch (error) {
      console.log(error.message)
      Alert.alert('Error', error.message);
    }
  };

  const payWithFirstPaymentMethod = async () => {
    try {
      if (paymentMethods.length > 0) {
        let firstPaymentMethod = paymentMethods[0];
        let result = await FiservSDKAPI.paymentConfirmProcessed(
          qrData.merchantInformation.merchantQRID,
          qrScannedTime,
          firstPaymentMethod.cardholder,
          firstPaymentMethod.maskedCardNumber,
          "C",
          firstPaymentMethod.issuerCode);   //Confirma el pago empleando los datos de dicho medio de
pago. Los medios de pagos procesados por Fiserv son del tipo "C" -> Crédito.
        console.log(result);
      } else {
        console.error("There are no allowed payment methods");
        Alert.alert('Error', "There are no allowed payment methods");
      }
    } catch (error) {
      console.error(error.message);
      Alert.alert('Error', error.message);
    }
  };

  const getPaymentMethods = async () => {
    try {
```

```
      const result = await FiservSDKAPI.getPaymentMethods();
      console.log("json: ", result);
      setPaymentMethods(parseProcessedPaymentMethods(result).paymentMethods);
      console.log("object: ", JSON.stringify(parseProcessedPaymentMethods(result), null, 4));
    } catch (error) {
      console.error(error.message);
      Alert.alert('Error', error.message);
    }
  };

  const handleQR = async () => {
    try {
      const result = await FiservSDKAPI.decodeQRCodeJson(

'0001140740013uy.com.urutec011221345257001202010032700000000000000000000000000099011415500 20uy.com.urutec.fiserv011221154230001802111277000344753038585405 10.045802UY5906FISERV6240030885678524070887654321101221386547001263 0485CB'
      );

      console.log("QRContent", result);

      // Decodifica el string de QR a un objeto Json con el modelo de datos del QR EMV
      const newQrData = new EMVCoQRData(JSON.parse(result));
      setQRData(newQrData);

       // Obtén la fecha y hora
      const qrScannedTime = await FiservSDKAPI.getCurrentDateTime();
      setQRScannedTime(qrScannedTime);
      console.log("Scanned at date/time: ", qrScannedTime);

      // Verifica que paymentMethods sea un array antes de aplicar .map
      if (Array.isArray(paymentMethods)) {
        const currentIssuerCodes = paymentMethods.map(method => method.issuerCode); // Obtén la lista de issuerCodes
        // Filtra los métodos de pago aceptados por el comercio
        const allowedIssuerCodes = await FiservSDKAPI.filterAllowedPaymentMethods(
          newQrData.merchantInformation.merchantQRID,
          newQrData.additionalData.merchantTaxID,
          qrScannedTime,
          currentIssuerCodes
        );
        console.log("issuerCodes: ", allowedIssuerCodes);
        const allowedPaymentMethods = paymentMethods.filter(method => allowedIssuerCodes.includes(method.issuerCode));

        // Actualiza el estado de paymentMethods
        setPaymentMethods(allowedPaymentMethods);

        console.log("Scanned at date/time: ", qrScannedTime);
        Alert.alert('Resultado del QR', result);
      } else {
        throw new Error('paymentMethods no es un array');
      }
    } catch (error) {
      console.error(error.message);
      Alert.alert('Error', error.message);
    }
  };

  return (
```

```
    <View style={styles.container}>
      <Text>Integración de Wallet QR Checkout</Text>
      <View style={{ height: 16 }} />
      <Button title="Iniciar Checkout" onPress={handleCheckout} />
      <View style={{ height: 16 }} />
      <Button
        title="Iniciar Pago sin UI"
        onPress={payWithFirstPaymentMethod}
        disabled={!qrData}
      />
      <View style={{ height: 16 }} />
      <Button title="Decodificar QR" onPress={handleQR} />
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
});
```