

SmartPOS API 1.5

Table of Contents

- [Table of Contents](#)
- [1 Overview](#)
 - [1.1 What's New](#)
 - [1.2 Revision History](#)
- [2 Getting Started](#)
- [3 Interface Structure](#)
 - [3.1 SmartPOS Info](#)
 - [Methods](#)
 - [3.2 Security](#)
 - [Methods](#)
 - [3.3 Printer](#)
 - [Methods](#)
 - [Print Objects](#)
 - [3.4 NFC & ChipCard Reader](#)
 - [3.4.1 NFC Methods](#)
 - [3.4.2 Chip Card Methods](#)
 - [Methods](#)
 - [3.5 Feature Check](#)
 - [Methods](#)
 - [3.6 Permissions](#)
 - [3.7 Service Connection](#)
- [4 Glossary](#)
 - [4.1 Troubleshooting](#)

1 Overview

Welcome to the developer documentation for the SmartPOS API, a powerful library that enables seamless integration with a wide range of smart point-of-sale (SmartPOS) devices. The SmartPOS API provides a unified interface to access various features and services provided by all Worldline offered SmartPOS devices.

The SmartPOS API is designed to facilitate communication and interaction with SmartPOS devices, allowing developers to leverage their capabilities for tasks such as compliant secure connections or access to the hardware features. By utilizing the library, you can access a range of functionalities tailored to meet the specific requirements of your application.

In this documentation, you will find detailed information on how to integrate and utilize each part of the SmartPOS API, along with code examples, best practices, and troubleshooting tips. We are excited to have you on board and look forward to helping you leverage the full potential of SmartPOS devices through the SmartPOS API. Let's get started!

1.1 What's New

- [2.3 Getting Started](#)
 - Example updated with new package name
- [3.3 Printer](#)
 - New method added: `getExtendedStatusOfPrinter()`
 - `getStatusOfPrinter()` deprecated

1.2 Revision History

- Revision #1
 - Document Creation

2 Getting Started

This guide will walk you through the steps to integrate the SmartPOS API into your Android Studio project. By following these instructions, you will be able to access the powerful features and functionality provided by SmartPOS devices within your application.

Prerequisites:

Before you begin, ensure that you have the following prerequisites in place:

1. **Android Studio:** Make sure you have the latest version of Android Studio installed on your development machine. You can download it from the official Android Studio website.
2. **SmartPOS API Library:** Obtain the SmartPOS API library from the official source.
3. **SmartPOS Device:** Ensure that you have a compatible SmartPOS device available for testing and integration.

Integration Steps:

Follow the steps below to integrate the SmartPOS API into your Android Studio project:

1. Create a New Android Studio Project:
 - Launch Android Studio and select "Create New Project" from the welcome screen.
 - Choose the desired project template and configure your project settings as needed.
 - Click "Finish" to create the project.
2. Import the SmartPOS API Library:
 - Copy the SmartPOS API library (.aar file) into your project's "libs" directory.
 - In Android Studio, Go to Project structure, Dependencies, click under 'All dependencies' +
 - Choose "JAR/AAR Dependency", type 'libs' under Step 1 and click OK.
 - Click Apply to automatically modify the build.gradle file from your project. The following code is now part of your project

Build.gradle.kts (:app)

```
...
dependencies {
    implementation(fileTree(mapOf(
        "dir" to "libs",
        "include" to listOf("*.aar", "*.jar"),
    )))
}
...
```

3. (OTHER OPTION) Configure the Module Dependencies:
 - a. Host the .AAR in your mavenLocal or Repository manager
 - b. Open your project's "build.gradle" file.

In the `dependencies` block, add the following line to include the SmartPOS API library as a dependency:

Build.gradle.kts (:app)

```
...
dependencies {
    implementation("com.worldline.smartposapi:core:<version>")
}
...
```

4. Sync Gradle and Build the Project:
 - Sync the Gradle files by clicking the "Sync Now" button that appears in the toolbar or by selecting "File" > "Sync Project with Gradle Files".
 - After the sync is complete, build the project by selecting "Build" > "Make Project" from the menu.
5. Start Using the SmartPOS API:
 - Now that the SmartPOS API library is integrated into your project, you can start utilizing its features.
 - Refer to the SmartPOS API documentation for specific instructions on how to use the different blocks, such as SmartPOS Info, Security, Printer, NFC & ChipCard Reader, and Feature Check.
 - Typically, you will need to instantiate the appropriate classes or interfaces provided by the library and call their methods to interact with the SmartPOS device.

Congratulations! You have successfully integrated the SmartPOS API into your Android Studio project. You can now explore the various functionalities offered by the library and leverage the power of SmartPOS devices within your application.

As the SmartPOS API is an AIDL interface we recommend you to read the official documentation to connect to out service at <https://developer.android.com/guide/components/aidl> . The service connection should be open only as long as your application needs to use each specific component, having a permanent open connection could result in performance issues.

Remember to consult the SmartPOS API documentation for detailed information on the available methods, parameters, and usage examples. If you encounter any issues during the integration process, refer to the troubleshooting section or reach out to the SmartPOS API support team for assistance.

Happy coding with SmartPOS devices and the SmartPOS API!

3 Interface Structure

The SmartPOS API provides a set of interfaces and methods to interact with SmartPOS devices and access their features. This API reference provides detailed documentation for each block of the SmartPOS API, including the SmartPOS Info, Security, Printer, NFC, ChipCard Reader and Feature Check blocks. The API as such is an AIDL interface - the Service runs on any SmartPOS device Worldline offers.

There are two kind of permissions that are mandatory check [Permissions](#)

Note: Before using any API methods, ensure that you have properly integrated the SmartPOS API library into your Android Studio project as described in the "Getting Started" section.

Name	Description
SmartPOS Info	The SmartPOS Info interface provides access to generic informal data on the used SmartPOS device. This information can be utilized for analysis, reporting, or customizing the user experience based on device-specific characteristics.
Security	The Security interface focuses on establishing secure connections with SmartPOS devices, utilizing compliant cypher suites to ensure the confidentiality and integrity of data exchanged between the device and your cloud connection.
Printer	The Printer interface enables interaction with the hardware printer integrated into the SmartPOS device. Through this block, you can initiate printing tasks, such as receipts or any other relevant documents, enhancing the transactional experience for your users.
NFC reader	The NFC Reader interface allows reading and writing to MIFARE and FELICA chip cards using Near Field Communication (NFC) technology. With this functionality, you can perform various operations on chip cards.
ChipCard Reader	ChipCard Reader interface allows reading of Belgium eID.
Feature Check	The Feature Check interface provides a mechanism to validate the supported features of the used SmartPOS device. This block helps you determine which specific features, such as printer or NFC functionality, are available on the device, allowing you to adapt your application's behavior accordingly.

By utilizing these different interfaces of the SmartPOS API, you can harness the power of SmartPOS devices and unlock a wide range of possibilities for your application. Whether you need to retrieve device information, establish secure connections, print receipts, interact with chip cards, or validate supported features, the SmartPOS API offers a comprehensive solution.

3.1 SmartPOS Info

The SmartPOS Info block provides methods to retrieve various information and statistics about the SmartPOS device. Use these methods to access data such as battery status, firmware version, network information, SIM card details, CPU stats, memory stats, and more.

Methods

- `getBatteryStatus(): BatteryStatus:`
 - Retrieves the current battery status of the device, including the battery level and charging status.
 - Returns:
 - A `BatteryStatus` object containing the battery information.
- `getCPUStats(): CPUStats:`
 - Retrieves the statistics related to the device's CPU usage, including the current CPU load and amount of processes.
 - Returns:
 - A `CPUStats` object containing the CPU statistics.
- `getDeviceMode(): DeviceMode:`
 - Retrieves the current mode of the SmartPOS device, indicating whether it is in production or development mode.
 - Returns:
 - A `DeviceMode` object representing the device mode.
- `getFirmwareVersion(): String:`
 - Retrieves the firmware version of the SmartPOS device.
 - Returns:
 - A string representing the firmware version.
- `getMemoryStats(): MemoryStats:`
 - Retrieves the statistics related to the device's memory usage, including the total available memory and the amount of free memory.
 - Returns:
 - A `MemoryStats` object containing the memory statistics.
- `getNetworkInfo(): NetworkInfo:`
 - Retrieves the network information of the SmartPOS device, including the mac address, available networks and active network
 - Returns:
 - A `NetworkInfo` object containing the network information.
- `getSerialNumber(): String:`
 - Retrieves the serial number of the SmartPOS device.
 - Returns:
 - A string representing the serial number.
- `getSimData(): MutableList<Sim>:`

- Retrieves the SIM card data of the SmartPOS device, including information about IMEI, Slot ID, APNs.
 - Returns:
 - A mutable list of `Sim` objects representing the SIM card data.
9. `getSimSlotIds(): MutableList<String>:`
- Retrieves the IDs of the SIM card slots available on the SmartPOS device.
 - Returns:
 - A mutable list of strings representing the SIM card slot IDs.
10. `selectSimApn(slotId: String?, apn: Apn?): ApnStatus:`
- Selects the specified Access Point Name (APN) configuration for the SIM card in the given slot.
 - Parameters:
 - `slotId`: The ID of the SIM card slot for which to select the APN configuration.
 - `apn`: The APN configuration to be used
 - Returns:
 - An `ApnStatus` object indicating the status of the APN selection.
11. `deleteSimApn(slotId: String?, apn: Apn?): ApnStatus:`
- Deletes the specified Access Point Name (APN) configuration from the SIM card in the given slot.
 - Parameters:
 - `slotId`: The ID of the SIM card slot from which to delete the APN configuration.
 - `apn`: The APN configuration to be deleted.
 - Returns:
 - An `ApnStatus` object indicating the status of the APN deletion.

3.2 Security

The Security block of the SmartPOS API provides methods related to creating secure connections with compliant cipher suites. Use these methods to request the supported ciphers for establishing secure communication.

Methods

1. `requestSupportedCiphers(): MutableList<String>:`
- Retrieves the list of supported cipher suites for secure connections.
 - Returns:
 - A mutable list of strings representing the supported cipher suites.

3.3 Printer

The Printer block of the SmartPOS API provides methods to interact with the hardware printer integrated into the SmartPOS device. Use these methods to perform various printing tasks, such as printing text, images, barcodes, and formatted lines.

Methods

1. `registerPrinterListener(printerListener: IPrinterListener)`
- Registers a listener to receive printer-related events and updates.
 - Parameters:
 - `printerListener`: An implementation of the `IPrinterListener` interface.
2. `unregisterPrinterListener(printerListener: IPrinterListener)`
- Unregisters a previously registered printer listener.
 - Parameters:
 - `printerListener`: The `IPrinterListener` implementation to be unregistered.
3. `getStatusOfPrinter(): PrinterStatus` (Deprecated, use `getExtendedStatusOfPrinter`)
- Retrieves the current status of the printer.
 - Returns:
 - `PrinterStatus` object indicating the current status of the printer.
4. `getExtendedStatusOfPrinter(): ExtendedPrinterStatus`
- Get the current status of the printer.
 - Returns:
 - `ExtendedPrinterStatus` object returning `ExtendedStatus` and `StatusCode`
 - `ExtendedStatus` object indicating the current status of the printer
 - `StatusCode` object indicating status code returned by device
5. `getMaxCharactersPerLine(printLineObject: PrintLineObject): Int`
- Retrieves the maximum number of characters per line supported by the printer for the given `PrintLineObject`.
 - Parameters:
 - `printLineObject`: The `PrintLineObject` containing the text to be printed.
 - Returns:
 - An integer representing the maximum number of characters per line.
6. `printDocument(document: List<PrintObject>): PrintResult`
- Handles the connection, the print request, and the closing of the connection when completed. This method takes a list of `PrintObject` instances and prints them accordingly.
 - Prints a list of different print objects based on the provided list. The print objects can represent various types, including lines, empty lines, barcodes, images, etc (see [Print Objects](#)).
 - Parameters:
 - `document`: A list of `PrintObject` instances, each representing a specific print object to be printed.

- Returns:
 - A `PrintResult` object indicating the result of the print operation.

Print Objects

The SmartPOS API provides various print objects that you can use to define your print content. The goal is to create a Document for printing all of them in a sequence without any interruption in the middle.

Each print object is designed to encapsulate specific types of content and printing layouts. Here are the available print objects along with their properties and explanations:

PrintLineObject

Represents a single line of text to be printed.

- Properties:
 - `data` (String): The text to be printed on one line.
 - `printLineProperties` (PrintLineProperties): Layout properties for the text to be printed. Optional.

PrintEmptyLines

Prints a specified number of empty lines with a given line height.

- Properties:
 - `amountOfEmptyLines` (Int): The number of empty lines to print.
 - `lineHeight` (Int): The height of each empty line in pixels.

PrintLeftRightInLine

Prints two lines of text, aligning the first line to the left and the second line to the right.

- Properties:
 - `printLineObjectLeft` (PrintLineObject): The `PrintLineObject` to be aligned on the left side of the line.
 - `printLineObjectRight` (PrintLineObject): The `PrintLineObject` to be aligned on the right side of the line.

PrintListOfLines

Prints a list of lines, each represented by a `PrintLineObject`.

- Properties:
 - `listOfLines` (List<PrintLineObject>): A list of `PrintLineObject` instances, each representing a single line to print.

PrintBarcodeObject

Prints a barcode using the provided barcode data and settings.

- Properties:
 - `contents` (String): The contents to be encoded to a barcode.
 - `barcodeFormat` (String): The barcode format that is supported by the printer.
 - `printBarcodeProperties` (PrintBarcodeProperties): Layout properties for the barcode. Optional.

PrintImageObject

Prints an image using the provided image data and settings.

- Properties:
 - `imageSource` (ImageSource): The source for the image to be printed.
 - `imageProperties` (ImageProperties): Layout properties for the image. Optional.

PrintLineProperties

Layout properties for text to be printed.

- Properties:
 - `fontSize` (Int): Font size following Microsoft Word's font size rule.
 - `typeface` (String): Typeface for the text.
 - `textAlignment` (String): Alignment of the text.
 - `isBold` (Boolean): Whether the text is bold.
 - `isItalic` (Boolean): Whether the text is italic.
 - `isUnderline` (Boolean): Whether the text is underlined.

PrintBarcodeProperties

Layout properties for the barcode.

- Properties:
 - `canvasHeight (Int)`: Canvas height in pixels.
 - `barcodeWidth (Int)`: Barcode width in pixels.
 - `barcodeHeight (Int)`: Barcode height in pixels.
 - `startDrawPositionX (Int)`: Start draw position on the canvas (horizontal).
 - `startDrawPositionY (Int)`: Start draw position on the canvas (vertical).

ImageSource

Information about the image source.

- Properties:
 - `imagePath (String)`: Path to the image on the device.
 - `imageContentProviderUrl (String)`: URI to the content provider that provides the image.
 - `imageByteArray (ByteArray)`: Bytearray content of the image.

ImageProperties

Layout properties for the image.

- Properties:
 - `canvasHeight (Int)`: Canvas height in pixels.
 - `imageWidth (Int)`: Image width in pixels.
 - `imageHeight (Int)`: Image height in pixels.
 - `startDrawPositionX (Int)`: Start draw position on the canvas (horizontal).
 - `startDrawPositionY (Int)`: Start draw position on the canvas (vertical).

3.4 NFC & ChipCard Reader

The NFC & ChipCard Reader block allows reading and writing to chip cards using Near Field Communication (NFC) technology. Use this block to perform various operations on chip cards, such as retrieving customer information, conducting transactions, or updating card data.

3.4.1 NFC Methods

- `readMifareSectors(mifareSectorRequest: MifareSectorRequest): CardReadResultMifareSector:`
 - Reads the specified Mifare sectors from a card using the provided `MifareSectorRequest`.
 - Parameters:
 - `mifareSectorRequest`: The `MifareSectorRequest` containing the details of the Mifare sectors to be read.
 - Returns:
 - A `CardReadResultMifareSector` object containing the result of the Mifare sector read operation.
- `readNfcData(auth: String?): CardReadResult:`
 - Reads NFC data from a card, optionally using authentication.
 - Parameters:
 - `auth`: An optional authentication string to authenticate the NFC data read operation.
 - Returns:
 - A `CardReadResult` object containing the result of the NFC data read operation.
- `readNfcUID(): CardReadResult:`
 - Reads the UID (Unique Identifier) from an NFC card.
 - Returns:
 - A `CardReadResult` object containing the result of the NFC UID read operation.
- `writeMifareSectors(mifareSectorRequest: MifareSectorRequest): CardReadResult:`
 - Writes data to the specified Mifare sectors on a card using the provided `MifareSectorRequest`.
 - Parameters:
 - `mifareSectorRequest`: The `MifareSectorRequest` containing the details of the Mifare sectors to be written.
 - Returns:
 - A `CardReadResult` object containing the result of the Mifare sector write operation.
- `writeNfcData(auth: String?, data: String?): CardReadResult:`
 - Writes data to an NFC card, optionally using authentication.
 - Parameters:
 - `auth`: An optional authentication string to authenticate the NFC data write operation.
 - `data`: The data to be written to the NFC card.
 - Returns:
 - A `CardReadResult` object containing the result of the NFC data write operation.

3.4.2 Chip Card Methods

The Chip Card Reader block of the SmartPOS API provides methods for interacting with chip cards. Please note that this version of SmartPOS API the chip card reader functionality is specifically designed for [Belgium eID](#) cards, only.

Methods

1. `readUID(): CardReadResult:`
 - Reads the UID (Unique Identifier) from a chip card.
 - Returns:
 - A `CardReadResult` object containing the result of the UID read operation.
2. `readEidData(isoCountryCode: String?): CardReadResult:`
 - Reads data from a Belgium eID card.
 - Parameters:
 - `isoCountryCode`: An optional ISO country code. For Belgium eID cards, this parameter can be omitted or set to "BE".
 - Returns:
 - A `CardReadResult` object containing the result of the eID data read operation.
3. `openCardReader(timeout: Int): CardOpenResult:`
 - Blocks until a card is inserted into the default card reader then attempts to power on the card and retrieve the ATR.
 - Parameters:
 - `timeout`: The maximum time to wait for a card to be inserted, in milliseconds.
Also, after a successful `openCardReader`, the exclusive access of the reader (for sending pages) will be given for that same length of time (and refreshed every call) to the calling AIDL Binder UID.
 - Returns:
 - A `CardOpenResult` object containing the ATR retrieved after powering on the card and the status of the card.
4. `sendAPDU(apdu: byte[], bufferSize: Int, closeAfterResponse: boolean): AduResponse:`
 - Sends an APDU command to the card and returns the response. Can but used multiple times in a row until a timeout happens.
 - Refreshes the ownership
 - Parameters:
 - `apdu`: The APDU command to be sent to the card.
 - `bufferSize`: Expected `bufferSize` of the response, use 255 per default.
 - `closeAfterResponse`: A flag indicating whether to close the card reader after receiving the response.
 - Returns:
 - The response of the card to the APDU command in an `AduResponse` object.
5. `closeCardReader():`
 - Closes the card reader.
 - Release the exclusive ownership of the card reader.

Here is a sample of how the open, send, and close apdu functions could be used.

Chaining open, send and close CardReader

```
// Open the card reader with a timeout of 5 seconds, acquiring exclusive ownership during this period
val openResult: CardOpenResult = chipCardReaderService.openCardReader(5000)

// Check the status of the card reader opening operation
if (openResult.status == CardReadStatus.SUCCESS) {
    val atr: String = openResult.atr // Extract ATR from the result
    // Process the ATR as needed

    // Example: Send an APDU command to the card, refreshing ownership
    val commandApu: ByteArray = byteArrayOf(0x00, 0xA4, 0x04, 0x00, 0x0A) // Example APDU command
    val bufferSize: Int = 255 // Set expected buffer size
    val closeAfterResponse: Boolean = true // Set whether to close the card reader after response

    // The sendAPDU function refreshes ownership during its execution
    val apduResponse: AduResponse = chipCardReaderService.sendAPDU(commandApu, bufferSize, closeAfterResponse)

    // Check the status of the APDU command execution
    when (apduResponse.status) {
        CardReadStatus.SUCCESS -> {
            val responseData: ByteArray = apduResponse.data // Extract response data from the result
            // Process the response data as needed
        }
        CardReadStatus.OPERATION_NOT_ALLOWED -> {
            // Handle the case where APDU command execution isn't allowed (APDU isn't whitelisted)
            // A possible action would be asking for a whitelist update or using a whitelisted apdu
            val errorMessage: String = apduResponse.errorMessage
            // Handle the error appropriately
        }
        else -> {
            // Handle other possible status values if needed
            val errorMessage: String = apduResponse.errorMessage
            // Handle the error appropriately
        }
    }

    // Close the card reader, releasing ownership
    chipCardReaderService.closeCardReader()
} else {
    // Handle the case where card reader opening failed
    val errorMessage: String = openResult.errorMessage
    // Handle the error appropriately
}
```

3.5 Feature Check

The Feature Check block of the SmartPOS API provides methods for validating the availability of specific features supported by the SmartPOS device. Use these methods to check the availability of various features before utilizing them in your application. Not all features are supported by every terminal. As example some devices has an integrated printer and some not. The one who doe not have a printer will not return the available feature "PRINTER".

Methods

1. `getAvailableFeatures(): MutableList<Feature>:`
 - Retrieves the list of available features supported by the SmartPOS device.
 - Returns:
 - A mutable list of `Feature` objects representing the available features.
2. `isFeatureAvailable(feature: Feature?): Boolean:`
 - Checks whether a specific feature is available on the SmartPOS device.
 - Parameters:
 - `feature`: The `Feature` to check for availability.
 - Returns:
 - A boolean value indicating whether the specified feature is available (`true`) or not (`false`).

3.6 Permissions

The SmartPOS API provides access to the NFC and ChipCard readers. To be able to access them, runtime permission is mandatory and they need to be defined into the Android Manifest.



Other interfaces do not need to request runtime permissions, if your application doesn't need to have access to NFC or ChipCard Reader, please skip this step.

AndroidManifest.xml

```
<uses-permission android:name="com.worldline.smartposapi.worldline.permission.NFC" />
<uses-permission android:name="com.worldline.smartposapi.worldline.permission.CHIP_CARD_READER" />
```

You also need to perform a runtime request for the user to grant these permissions.

Requesting permission in the Activity

```
class TestActivity : AppCompatActivity() {

    private val requestPermissionLauncher =
        registerForActivityResult(
            ActivityResultContracts.RequestMultiplePermissions(),
            ::onPermissionResult
        )

    override fun onCreate(savedInstanceState: Bundle?, persistentState: PersistableBundle?) {
        super.onCreate(savedInstanceState, persistentState)
        checkAndRequestPermission()
    }

    /**
     * Check if the user currently has the required permission and request the runtime permission
     * if necessary
     */
    private fun checkAndRequestPermission() {

        val hasNfc = ContextCompat.checkSelfPermission(
            this, Permissions.NFC
        ) == PackageManager.PERMISSION_GRANTED
        val hasChipCardReader = ContextCompat.checkSelfPermission(
            this, Permissions.CHIP_CARD_READER
        ) == PackageManager.PERMISSION_GRANTED

        val hasAllPermissions =
            hasNfc &&
                hasChipCardReader

        if (!hasAllPermissions) {
            requestPermissionLauncher.launch(
                arrayOf(
                    Permissions.NFC,
                    Permissions.CHIP_CARD_READER
                )
            )
        }
    }

    /**
     * Handle the permission result of the launched permission request
     */
    private fun onPermissionResult(permissionMap: Map<String, Boolean>) {
        val refusedPermissions = permissionMap.filter { (_, v) -> !v }
        if (refusedPermissions.isEmpty()) {
            //all permissions were granted, do something
        } else {
            //some of the requested permissions were not granted
        }
    }
}
```

3.7 Service Connection

As we mentioned before, SmartPOS API is an AIDL interface. In order to connect to the different interfaces available you must establish a connection for each service that you would like to use. This is an example of how you can establish service connection based on the documentation of Android that can be found here; <https://developer.android.com/guide/components/aidl#Calling>

The following demonstrates how to establish connection with the FeatureCheck interface and print the available features of the terminal when we click a button :

- Keep in mind that the way to connect to the service needs to change depending on the Interface you are using.

MainActivity.kt

```
class MainActivity : ComponentActivity() {

    companion object {
        private const val DEFAULT_SMARTPOS_API_PACKAGE = "com.worldline.smartposapi.service"
    }

    var iFeatureCheckService: IFeatureCheckService? = null

    private val requestPermissionLauncher =
        registerForActivityResult(
            ActivityResultContracts.RequestMultiplePermissions(),
            ::onPermissionResult
        )

    private val serviceConnectionFeatureCheck = object : ServiceConnection {
        override fun onServiceConnected(className: ComponentName?, service: IBinder?) {
            iFeatureCheckService = IFeatureCheckService.Stub.asInterface(service)
        }

        override fun onServiceDisconnected(name: ComponentName?) {
            iFeatureCheckService = null
            Toast.makeText(
                applicationContext,
                "Service has unexpectedly disconnected",
                Toast.LENGTH_SHORT
            ).show()
        }
    }

    private fun bindToFeatureCheck() {
        try {
            val secondIntent = Intent(BindingActions.FEATURE_CHECK)
            secondIntent.putExtra(BindingExtras.API_VERSION, "v1")
            secondIntent.setPackage(DEFAULT_SMARTPOS_API_PACKAGE)
            bindService(
                secondIntent,
                serviceConnectionFeatureCheck,
                Context.BIND_AUTO_CREATE
            ).also {
                Log.d("MainActivity", "Bind Feature Check : $it")
            }
        } catch (e: Exception) {
            Log.d("Main Activity", "not able to bind ! ")
        }
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView {
            SampleSmartPosApiTheme {
                // A surface container using the 'background' color from the theme
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    Column(
                        modifier = Modifier.fillMaxWidth(),
                        verticalArrangement = Arrangement.Center
                    ) {
                        CheckAvailableFeatures()
                    }
                }
            }
        }
        checkAndRequestPermission()
    }

    override fun onStart() {
```

```

        super.onStart()
        bindToFeatureCheck()
        Log.d("MainActivity", "onStart bind service")
    }

    override fun onStop() {
        super.onStop()
        unbindService(serviceConnectionFeatureCheck)
        Log.d("MainActivity", "onStop unbind service")
    }

    @Composable
    fun CheckAvailableFeatures(modifier: Modifier = Modifier) {
        var showInfo by remember { mutableStateOf(false) }
        Column(
            modifier = modifier
                .fillMaxWidth()
                .padding(12.dp),
            verticalArrangement = Arrangement.Center,
            horizontalAlignment = Alignment.CenterHorizontally
        ) {
            Button(
                onClick = {
                    showInfo = !showInfo
                },
                modifier = modifier
            ) {
                Text(text = "Check Available Features")
            }
            if (showInfo) {
                Text(text = "Available Features: \n ${iFeatureCheckService?.availableFeatures}")
            }
        }
    }

    /**
     * Check if the user currently has the required permission and request the runtime permission
     * if necessary
     */
    private fun checkAndRequestPermission() {
        val hasNfc = ContextCompat.checkSelfPermission(
            this, Permissions.NFC
        ) == PackageManager.PERMISSION_GRANTED
        val hasChipCardReader = ContextCompat.checkSelfPermission(
            this, Permissions.CHIP_CARD_READER
        ) == PackageManager.PERMISSION_GRANTED

        if (!(hasNfc && hasChipCardReader)) {
            requestPermissionLauncher.launch(
                arrayOf(
                    Permissions.NFC,
                    Permissions.CHIP_CARD_READER
                )
            )
        }
    }

    /**
     * Handle the permission result of the launched permission request
     */
    private fun onPermissionResult(permissionMap: Map<String, Boolean>) {
        val refusedPermissions = permissionMap.filter { (_, v) -> !v }
        if (refusedPermissions.isEmpty()) {
            //all permissions were granted, do something
        } else {
            //some of the requested permissions were not granted
        }
    }
}

```

4 Glossary

4.1 Troubleshooting

During the integration and usage of the SmartPOS API, you may encounter certain issues or challenges. This section provides some common troubleshooting steps to help you resolve them.

1. **Connection Issues:** If you are experiencing connection problems with the SmartPOS device, try the following steps:
 - Ensure that the SmartPOS device is powered on and within range.
 - Verify that the device address or connection parameters are correct.
 - Check the WIFI / ethernet settings on both the SmartPOS device and your device.
 - Restart both the SmartPOS device and your device.
2. **API Initialization:** If you are facing issues during the initialization of the SmartPOS API, consider the following:
 - Double-check that you have imported the correct classes from the SmartPOS API library.
 - Ensure that you have connected to the `SmartPOS` service properly.
 - Verify that the necessary permissions are declared in your Android manifest file.
3. **Functional Issues:** If you are encountering problems with specific API functionalities, follow these steps:
 - Review the documentation and ensure that you are using the correct methods and parameters.
 - Check for any required permissions or additional configurations for the specific functionality.
 - Validate that the SmartPOS device supports the particular feature you are trying to utilize.
 - Ensure that you are handling the API responses and errors appropriately in your code.
4. **Device Compatibility:** If you are unsure about the compatibility of your SmartPOS device with the API, consider the following:
 - Refer to the SmartPOS API documentation or contact the SmartPOS support team to confirm device compatibility.
 - Check for any specific requirements or limitations related to your SmartPOS device model.
5. **Error Messages and Logs:** Pay attention to any error messages or logs provided by the API or the SmartPOS device. These can provide valuable insights into the cause of the issue. Make sure to capture and analyze them for troubleshooting purposes.

If you have exhausted all troubleshooting steps and are still unable to resolve the issue, consider reaching out to the SmartPOS support team for further assistance. They can provide specific guidance tailored to your integration and help you overcome any challenges you may be facing.

Remember to keep your SmartPOS API version and firmware up to date to ensure compatibility and access to the latest features and bug fixes.