



HF Reader App — API Documentation

Scope: Mobile internal APIs (TypeScript) and native bridge for NFC operations; Redux slices used by UI flows.

Version: 0.0.1

Last updated: 2025-08-10

iDTRONIC GmbH

OVERVIEW AND SCOPE

This document describes the programmatic interfaces available inside the HF Reader mobile application. It focuses on:

- Enhanced NFC Reader API (read pipeline, data contracts, errors)
- Batch NFC Writer API (record construction, write, progress, errors)
- NFC data types (card families, NDEF structures, security/memory details)
- Android native bridge (NfcExtras: ATQA/SAK utilities)
- Redux slices (Theme, SavedTags, Queue, Auth) with actions/selectors

This is not a public HTTP API; it documents in-app modules and their contracts for developers extending screens, building automation, or integrating vendor-specific tag capabilities and diagnostics.

Platforms and minimums:

- Android: NFC hardware required; tech filters for NfcA, NfcB, NfcF, NfcV, MifareClassic, MifareUltralight, IsoDep, Ndef, NdefFormatable.
- iOS: iOS 11+ NFC-capable devices; NDEF/TAG reader session entitlements.

TECHNOLOGY AND NAMING

Language: TypeScript (React Native 0.79.2, React 19)

State: Redux Toolkit + redux-persist (AsyncStorage)

Navigation: @react-navigation/native and native-stack

NFC: react-native nfc-manager; Android native additions in Kotlin

Module path notes:

- Services reside under app/services/
- Enhanced NFC reader under app/services/enhanced/
- Slices under app/redux/slice/
- Store under app/redux/store/

ENHANCED NFC READER API

Location: `app/services/enhanced/NFCReaderManager.ts`

Description A singleton service that orchestrates NFC reads. It negotiates across several technologies, detects card families, reads low-level and NDEF content, performs security analysis, and returns a normalized `NFCCardInfo` object. It supports retries, timeouts, and structured errors.

Instantiation `NFCReaderManager.getInstance(): NFCReaderManager`

Configuration (`NFCReaderConfig`)

- `retryCount`: number (default: 2)
- `timeout`: number in ms (default: 20000)
- `enableLogging`: boolean (default: true)
- `autoRetry`: boolean (default: true)
- `preferredTechnology?`: string (one of `NfcTech.*` if you want to hint a tech)
- `authenticationKeys?`: string[] (default MIFARE keys catalog)
- `skipAuthentication?`: boolean (skip auth probing where applicable)

Public Methods

- `initialize()`: Promise Starts NfcManager. Throws a typed error if unsupported or failed.
- `readCard(config?: Partial)`: Promise Attempts to connect using the following order: NfcA, NfcB, NfcF, NfcV, IsoDep, MifareClassic, MifareUltralight, Ndef On success, returns `success:true` with a `NFCCardInfo` payload containing card type, UID, technologies, memory/security information, and parsed NDEF. Respects `retryCount/autoRetry` and `timeout`. Cleans up the NFC session.
- `isEnabled()`: Promise Returns whether device NFC is currently enabled.
- `stop()`: Promise Cancels active technology requests and stops the manager.

Return Types

- `CardReaderResult` (union): `success: true, cardInfo: NFCCardInfo, retryCount?: number` OR `success: false, error: string, retryCount?: number`

Error Model (subset) `NFCErrorType` = `INITIALIZATION_FAILED` | `TAG_CONNECTION_FAILED` | `AUTHENTICATION_FAILED` | `READ_ERROR` | `WRITE_ERROR` | `UNSUPPORTED_CARD` | `TIMEOUT` | `USER_CANCELLED` | `UNKNOWN_ERROR`

Typical Usage

```
const reader = NFCReaderManager.getInstance(); const result = await reader.readCard({ timeout: 15000, autoRetry: true }); if (result.success) { // result.cardInfo.technicalDetails.uid, etc. } else { // Show error; consider retry depending on error type }
```

Notes and Behavior

- Concurrent reads are prevented; attempting to read while in progress returns a `READ_ERROR` with a clear message.
- Certain error types (`USER_CANCELLED`, `UNSUPPORTED_CARD`, `AUTHENTICATION_FAILED`) disable auto-retry for efficiency.
- Cleanup is guaranteed in finally blocks.

BATCH NFC WRITER API

Location: `app/services/BatchNFCService.ts`

Description Writes one or more NDEF records to an engaged NFC tag. It reports progress via a callback, normalizes and formats common URI schemes, and emits clear user-facing messages for typical failure modes (timeouts, IO errors, non-writable tags, insufficient memory, unsupported APIs).

Public Methods

- `initialize()`: Promise Idempotent initialization of `NfcManager`.
- `writeBatchToTag(records: QueuedRecord[], onProgress?: (percent:number)=>void)`: Promise Steps: 1) Ensures initialization; requests Ndef technology. 2) Heuristically checks writability from `tag.techTypes` (if available). 3) Creates NDEF records from each `QueuedRecord`, handling transforms. 4) Encodes message (`Ndef.encodeMessage`) and writes via `ndefHandler`. 5) Reports progress: ~0–50% during record construction; ~50–100% for encode/write. Returns true on success; returns false and shows a message on recoverable errors (user cancel, IO, timeout, not writable, tag lost, insufficient memory).
- `stopNFC()`: Promise Cancels technology request and resets internal state.

QueuedRecord (from app/redux/slice/queueSlice.tsx)

- id: string (generated by slice; omit on creation)
- title: string
- description: string
- data: string (raw input; transformation depends on transformType)
- recordType: 'text' | 'url' | 'custom'
- transformType?: 'email' | 'phone' | 'url' | 'bitcoin' | 'search' | 'facetime' | 'facetime audio' | 'bluetooth' | 'sms' | 'location' | 'address' | 'contact' | 'wifi' | 'none'
- customPayload?: string (for MIME media)
- customMimeType?: string (e.g., 'text/vcard')
- timestamp: number (generated by slice; omit on creation)

Transform Handling (examples)

- 'url': ensure scheme; prepend https:// if missing
- 'email': mailto:
- 'phone': tel:
- 'sms': sms:?body=
- 'location': geo:,
- 'search': https://www.google.com/search?q=
- 'facetime' / 'facetime-audio': facetime: / facetime-audio:
- 'wifi': WIFI:S:\nT:\nP:;; • 'contact': vCard template from data string

Typical Usage `const wrote = await BatchNFCService.writeBatchToTag(queueRecords, p => setProgress(p)); if (!wrote) { /* prompt retry */ }`

NFC DATA TYPES (SELECTED)

Location: `app/services/enhanced/NFCTypes.ts`

CardType (enum) UNKNOWN NTAG_213, NTAG_215, NTAG_216, NTAG_424
MIFARE_CLASSIC_1K, MIFARE_CLASSIC_4K MIFARE_ULTRALIGHT, MIFARE_ULTRALIGHT_C,
MIFARE_ULTRALIGHT_EV1 MIFARE_DESFIRE, MIFARE_DESFIRE_EV1, MIFARE_DESFIRE_EV2
ISO15693, FELICA

NDEFRecord

- recordType: string
- typeNameFormat: number
- id?: string
- payload: string
- language?: string
- encoding?: string
- uri?: string
- mimeType?: string

NDEFContent

- messageSize: number
- writeable: boolean
- canBeMadeReadOnly: boolean
- records: NDEFRecord[]

MemoryInfo

- totalSize: number
- pageSize: number
- userMemoryStart: number
- userMemoryEnd: number
- readOnly: boolean[]
- reserved: boolean[]
- lockable: boolean[]
- locked: boolean[]

SecurityInfo

- isProtected: boolean
- authenticationRequired: boolean
- writeProtected: boolean
- hasDefaultKeys: boolean
- supportedAuthMethods: string[]
- securityStatus: { [sector: string]: { readable: boolean, writable: boolean, authenticatedWith?: string } }

TechnicalDetails (subset)

- uid: string
- atqa?: string
- sak?: string
- historicalBytes?: string[]
- protocolInfo?: string
- applicationData?: string
- manufacturer: string
- productType?: string
- hardwareVersion?: string
- softwareVersion?: string

NFCCardInfo

- cardType: CardType
- uid: string
- technologies: string[]
- memoryInfo: MemoryInfo
- rawData: { [blockOrPage: string]: { data: number[], ascii: string } }
- ndefContent?: NDEFContent
- securityInfo: SecurityInfo
- technicalDetails: TechnicalDetails
- errorMessages: string[]
- lastReadTime: Date
- readDuration: number (ms)

NFCErrorType (enum) INITIALIZATION_FAILED, TAG_CONNECTION_FAILED, AUTHENTICATION_FAILED, READ_ERROR, WRITE_ERROR, UNSUPPORTED_CARD, TIMEOUT, USER_CANCELLED, UNKNOWN_ERROR

ANDROID NATIVE MODULE — NFCEXTRAS

Location: android/app/src/main/java/com/hfreaderapp/nfc/NfcExtrasModule.kt

Module name: "NfcExtras"

Methods

- `isAvailable()`: Promise Returns whether `android.hardware.nfc` is present.
- `getAtqaSakFromIntent()`: `Promise<{ atqaHex:string, sakHex:string, atqa:[number,number], sak:number } | null>` If current Activity was launched via NFC intent, extracts the Tag and returns ATQA/SAK as numbers and hex strings; null if not present.
- `readAtqaSakWithReaderMode(timeoutMs: number)`: `Promise<{ atqaHex:string, sakHex:string, atqa:[number,number], sak:number } | null>` Enables reader mode for NFC-A (skip NDEF check), resolves once upon discovery or after timeout. Serialized access; always disables reader mode.

Behavior notes

- Concurrency guarded via a `readerActive` flag and single pending promise.
- Main thread handler is used to schedule timeout and UI-thread calls.

INTEGRATION EXAMPLES

Read and persist a tag (pseudo-flow) `const res = await`

```
NFCReaderManager.getInstance().readCard({ timeout: 15000 }); if (res.success &&
res.cardInfo) { const info = res.cardInfo; dispatch(saveTag({ tagType: info.cardType,
technologies: info.technologies.join(','), serialNumber: info.technicalDetails.uid, atqa:
info.technicalDetails.atqa ?? '', sak: info.technicalDetails.sak ?? '', protected:
info.securityInfo.writeProtected, memoryInfo: String(info.memoryInfo.totalSize),
dataFormat: 'NDEF', size: String(info.ndefContent?.messageSize ?? 0), writable:
Boolean(info.ndefContent?.writable), ndefRecords: info.ndefContent?.records ?? [],
rawData: JSON.stringify(info.rawData), name: undefined, })); }
```

Write queued records to a tag (pseudo-flow) `const ok = await`

```
BatchNFCService.writeBatchToTag(queue.records, p => dispatch(setWriteProgress(p)) ); if
(!ok) { /* show retry UI */ }
```

ERROR HANDLING AND UX GUIDANCE

Reader

- Avoid parallel reads; show a busy indication when in progress.
- For TIMEOUT or IO-like READ_ERROR, suggest holding device steady and retrying.
- For USER_CANCELLED, no retry suggestion necessary.

Writer

- Tag not writable: instruct users to use a compatible NDEF tag.
- Insufficient memory: reduce number/size of records, or choose a larger tag.
- Tag lost / connection failed: keep device and tag closely coupled; retry.
- Unsupported tag API: try a different tag type or device.

SECURITY AND PRIVACY

- NFC content is processed locally. No network transmission by default.
- Persisted SavedTags are stored in AsyncStorage (device storage).
- Ensure platform permission prompts explain purpose (NFC/camera/mic where used).
- For any future export/sync feature, use encryption in transit and at rest, with explicit user consent and updated privacy disclosures

**For Inquiries,
Contact Us**

