Developer API & Integrations

- Overview · Developer API & Integrations
- Integration API Overview
- Canonical Item IDs & Aliases
- Server Exports · Catalog & Opening
- Server Exports · Add, Remove & Carry
- Server Exports · Read & Search
- Server Exports · Metadata & Inventory Management
- Client Read-Only Exports
- Registering Stashes & Custom Inventories
- Usable Items & External Handlers
- Developer Ecosystem SDK
- Categories & Runtime Catalogs
- Integration Manifests & Provider Ownership
- Item Actions, Metadata Schemas & Images
- Durability & Container API
- Hooks, Capabilities & Diagnostics
- Admin Access Exports
- Player Interaction Exports
- Notify & Text UI Exports
- OX / QB Migration Examples
- Complete Export Index
- Operations API · v2.0.0

Overview · Developer API & Integrations

4 · Developer API & Integrations

Complete public export surface, runtime registration, provider ownership and migration examples.

v2.0.0 adds four trusted server-side Operations/Forensics exports.

Integration API Overview

Integration API Overview

Public, versioned integration surface for trusted FiveM resources.

Applies to: `Nord Inventory v2.0.0`

Nord Inventory — Integration API

Nord Inventory remains a Nord-native inventory. The public API intentionally uses familiar slot-inventory contracts so existing FiveM resources need minimal inventory-specific code.

Typical migration

A resource that reads an item count can use:

```
local count = exports.nord_inventory:GetItemCount(source, 'water')
```

Slot lookup:

```
local slot = exports.nord_inventory:GetSlotWithItem(source, 'water', { type = 'clean' })
```

Search:

```
local count = exports.nord_inventory:Search(source, 'count', 'water')
local slots = exports.nord_inventory:Search(source, 'slots', 'water')
```

The same read-only search helpers are also available client-side for the local player's inventory:

```
local count = exports.nord_inventory:Search('count', 'water')
local items = exports.nord_inventory:GetPlayerItems()
```

UI bridge exports

Notifications and world prompts are also exposed as a stable Nord gateway. The visual resources are selected only by name in `Config.Notify`, `Config.TextUI` and `Config.Target`; consuming resources do not need to know which notification/TextUI resource is installed.

Server:

- `Notify(source, message, kind?, duration?, title?)`
- `ShowTextUI(source, text, options?)`
- `HideTextUI(source)`
- `ShowTextUIEntry(source, id, data)`
- `HideTextUIEntry(source, id)`

Client:

- `Notify(message, kind?, duration?, title?)`
- `ShowTextUI(text, options?)`
- `HideTextUI()`
- `ShowTextUIEntry(id, data)`
- `HideTextUIEntry(id)`

See `docs/UI_BRIDGE.md` for the minimal resource-name configuration and bridge contracts.

Server exports

Core catalogue and registry:

- `GetInventoryType()`
- `GetItemDefinition(name)`
- `GetItems()`
- `GetItemList()`
- `Items(itemName?)`
- `ItemList(itemName?)`
- `RegisterItem(name, definition, useHandler?, side?)`
- `RegisterItems(definitions)`
- `RegisterInventory(id, options)`
- `RegisterStash(id, label, slots, maxWeight, owner?, groups?, coords?)`

Inventory operations:

- `AddItem(inv, item, count, metadata?, slot?, cb?)`
- `RemoveItem(inv, item, count, metadata?, slot?, ignoreTotal?, strict?)`
- `CanCarryItem(inv, item, count, metadata?)`
- `CanCarryAmount(inv, item)`
- `CanCarryWeight(inv, weight)`
- `CanSwapItem(inv, firstItem, firstCount, testItem, testCount)`
- `ClearInventory(inv, keep?)`
- `RemoveInventory(inv)`
- `SetMaxWeight(inv, maxWeight)`
- `SetSlotCount(inv, slots)`

Read/search helpers:

- `GetInventory(inv, owner?)`
- `GetInventories(type?, detailed?)`
- `GetInventoryItems(inv, owner?)`
- `GetItem(inv, item, metadata?, returnsCount?)`
- `GetItemCount(inv, item, metadata?, strict?)`
- `GetItemSlots(inv, item, metadata?)`
- `GetSlot(inv, slot)`
- `GetSlotForItem(inv, item, metadata?)`
- `GetSlotWithItem(inv, item, metadata?, strict?)`
- `GetSlotsWithItem(inv, item, metadata?, strict?)`
- `GetSlotIdWithItem(inv, item, metadata?, strict?)`
- `GetSlotIdsWithItem(inv, item, metadata?, strict?)`
- `GetEmptySlot(inv)`
- `Search(inv, 'slots'|'count', item|string[], metadata?)`
- `SearchInventory(inv, 'slots'|'count', item|string[], metadata?)`

Metadata/use:

- `SetMetadata(inv, slot, metadata)`
- `GetMetadata(inv, slot)`
- `UpdateMetadata(inv, slot, patch)`
- `RegisterUsableItem(item, callback|export, side?)`
- `RegisterUseExport(item, export, side?)` / `RegisterUseExport(item, resource, export, side?)`
- `RegisterHook(name, callback)` / `registerHook(name, callback)`

Inventory identifiers

Player inventories use the server id (`source`). Registered stashes can be addressed by their registered id. Full Nord keys such as `stash:police_armory`, `trunk:ABC123`, and `glovebox:ABC123` are

also accepted when they exist.

Metadata strings are normalized to `{ type = value }`. The `strict` flag on slot/count helpers switches between exact metadata matching and partial matching.

Runtime item registration with an external use export

Trusted server resources can register an item definition without editing `shared/items.lua` and can declare its external use handler in the same definition. A bare `client.export` or `server.export` name is automatically bound to the resource that called `RegisterItem`.

```
local ok, def = exports.nord_inventory:RegisterItem('service_token', {
    label = 'Service Token',
    description = 'Issued by another resource',
    weight = 5,
    stack = true,
    close = false,
    category = 'document',
    schema = 'generic',
    image = 'service_token.png',
    metadataDefaults = {
        issuer = '',
        issuedAt = 0
    },

    -- OX-style external use handler. Because the export name is bare,
    -- Nord resolves it as <calling_resource>:UseServiceToken.
    client = {
        export = 'UseServiceToken',
        remove = 0
    }
})

exports('UseServiceToken', function(item, slot)
    -- Return false to cancel the use transaction.
    return true
end)
```

For a server-side handler, use `server` instead of `client`:

```
server = {
    export = 'UseServiceToken',
    remove = 1
}

exports('UseServiceToken', function(source, item, slot, definition)
    return true
end)
```

Important precedence rule

A runtime item **cannot overwrite an authoritative database row** with the same name. `RegisterItem` returns a failure such as `database_authoritative` in that case.

This protects Admin Studio/database definitions from being silently replaced by a third-party resource on restart.

Runtime lifetime

`RegisterItem` is transient. If you need a definition to be permanently managed by Nord Inventory, create/import it into the database-backed registry or maintain it in the reference Lua files.

External item use exports

Usable items can delegate their action to another resource while Nord remains responsible for inventory validation and optional removal. See `docs/EXTERNAL_ITEM_USE.md`.

```
-- Compact: UseRepairKit belongs to the resource calling this export.
exports.nord_inventory:RegisterUseExport('repair_kit', 'UseRepairKit', 'server')

-- Explicit resource form remains supported.
exports.nord_inventory:RegisterUseExport('repair_kit', 'my_mechanic', 'UseRepairKit',
'server')
```

Or register the definition and export in one call:

```
exports.nord_inventory:RegisterItem('repair_kit', {
    label = 'Repair Kit',
    weight = 500,
    use = {
        export = 'my_mechanic:UseRepairKit',
        side = 'server',
        remove = 1
    }
})
```

Internal durability repair items

A registered item can use Nord's built-in server-authoritative durability repair effect instead of an external use export:

```
exports.nord_inventory:RegisterItem('pistol_cleaning_kit', {
    label = 'Pistol Cleaning Kit',
    weight = 200,
    stack = true,
    close = true,
    use = {
        duration = 3000,
        remove = 1,
        repair = {
            item = 'weapon_pistol',
            amount = 30
        }
    }
})
```

`repair.item` is the internal name of the target definition and `repair.amount` is the number of durability percentage points restored. Nord repairs the most damaged matching instance, never exceeds the target maximum, and only consumes the repair item when a repair was actually applied.

Developer Ecosystem API

For provider-aware integrations, prefer the ecosystem APIs below. Full examples are in `docs/DEVELOPER_ECOSYSTEM.md`.

```
-- Feature detection / diagnostics
exports.nord_inventory:HasFeature('integration_manifest')
exports.nord_inventory:GetCapabilities()
exports.nord_inventory:GetRegisteredProvider('item_name')
exports.nord_inventory:GetProviderRegistry()
exports.nord_inventory:DebugItem('item_name')
exports.nord_inventory:GetHealthReport()

-- Full provider manifest
exports.nord_inventory:RegisterIntegration({ ... })
exports.nord_inventory:UnregisterIntegration()

-- Runtime ownership
exports.nord_inventory:UnregisterItem('item_name')
exports.nord_inventory:UnregisterItems({ 'item_one', 'item_two' })
exports.nord_inventory:UnregisterUseHandler('item_name')

-- Context-menu actions
exports.nord_inventory:RegisterItemAction('item_name', { ... })
exports.nord_inventory:UnregisterItemAction('item_name', 'action_id')
exports.nord_inventory:GetItemActions('item_name')

-- Metadata extensions
exports.nord_inventory:RegisterMetadataSchema('item_name', schema, display)
exports.nord_inventory:UnregisterMetadataSchema('item_name')
exports.nord_inventory:GetMetadataSchema('item_name')

-- Provider item images
exports.nord_inventory:RegisterItemImage('path/in/provider.png', 'item_name.png')

-- Item containers
exports.nord_inventory:OpenItemContainer(source, slot)
exports.nord_inventory:GetContainerInventory(source, slot)

-- Durability
exports.nord_inventory:GetDurability(source, slot)
exports.nord_inventory:SetDurability(source, slot, value)
```

```
exports.nord_inventory:AddDurability(source, slot, amount)
exports.nord_inventory:RemoveDurability(source, slot, amount)
exports.nord_inventory:RepairItem(source, slot, amount)
```

All provider mutation APIs above are server-side. Provider ownership is resolved from the resource calling the export.

Hide player metadata details

```
exports['nord_inventory']:RegisterItem('example_item', {
    label = 'Example Item',
    weight = 100,
    hideMetadataDetails = true
})
```

`metadata.label` and `metadata.image` remain presentation fields and still affect the visible item name/image. Other metadata remains available to scripts but is not shown in the player's item-information metadata section.

Register category + items in one call

```
exports['nord_inventory']:RegisterCategoryItems({
    name = 'mechanic_materials',
    label = 'Mechanic Materials',
    icon = 'fa-screwdriver-wrench',
    order = 80
}, {
    mech_sandpaper = { label = 'Sandpaper', weight = 150 },
    mech_primer = { label = 'Paint Primer', weight = 500 },
    mech_clearcoat = { label = 'Clear Coat', weight = 500 }
})
```

All item definitions inherit `mechanic_materials` automatically.

For multiple folders in the same resource:

```
exports['nord_inventory']:RegisterItemCatalog({
    mechanic_paint = {
        label = 'Paint Materials',
```

```
        icon = 'fa-spray-can-sparkles',
        items = {
            mech_basecoat = { label = 'Base Coat', weight = 500 }
        }
    },
    mechanic_tools = {
        label = 'Mechanic Tools',
        icon = 'fa-screwdriver-wrench',
        items = {
            mech_wrench = { label = 'Workshop Wrench', weight = 900 }
        }
    }
})
```

Register categories

```
exports['nord_inventory']:RegisterCategory('restaurant_food', {
    label = 'Restaurant Food',
    icon = 'fa-utensils',
    order = 100
})
```

`RegisterCategory` is idempotent. Existing categories are reused.

A `RegisterIntegration` manifest can install `categories = { ... }` before its `items = { ... }`, or an individual `RegisterItem` can use an inline category table containing `name`, `label`, `icon`, and `order`.

Canonical Item IDs & Aliases

Canonical Item IDs & Aliases

Keep one item identity while supporting legacy third-party names.

Applies to: `Nord Inventory v2.0.0`

Nord Inventory resolves legacy aliases before authoritative item operations. The default alias map includes common QBCore-style ammunition IDs such as `pistol_ammo`, `rifle_ammo`, `shotgun_ammo`, etc.

```
local canonical = exports.nord_inventory:ResolveItemName('pistol_ammo')
-- -> ammo-9 with the default alias table
```

Add server-specific legacy names to `Config.ExternalItemAliases` instead of duplicating item definitions under multiple names.

Server Exports · Catalog & Opening

Server Exports · Catalog & Opening

Catalog lookup, item identity and inventory opening.

Applies to: `Nord Inventory v2.0.0`

Common exports:

```
exports.nord_inventory:GetInventoryType()
exports.nord_inventory:GetItemDefinition(name)
exports.nord_inventory:GetItems()
exports.nord_inventory:GetItemList()
exports.nord_inventory:ResolveItemName(value)
exports.nord_inventory:GetExternalItemDefinition(value)
exports.nord_inventory:GetItemCatalog()
exports.nord_inventory:GetItemImage(value, mode)
exports.nord_inventory:GetItemImageData(value)
exports.nord_inventory:OpenInventory(target, invType, data)
exports.nord_inventory:openInventory(target, invType, data)
exports.nord_inventory:forceOpenInventory(target, invType, data)
```

Use canonical item names for persistence and external aliases only at the boundary of third-party compatibility.

Server Exports · Add, Remove & Carry

Server Exports · Add, Remove & Carry

Safe item/money mutations and preflight capacity checks.

Applies to: `Nord Inventory v2.0.0`

```
local ok, result = exports.nord_inventory:AddItem(source, 'water', 2, { quality = 100 })
local removed = exports.nord_inventory:RemoveItem(source, 'water', 1)
local canCarry = exports.nord_inventory:CanCarryItem(source, 'water', 5)
local remaining = exports.nord_inventory:CanCarryAmount(source, 'water')
```

Available money helpers:

```
exports.nord_inventory:AddCash(target, amount, metadata)
exports.nord_inventory:RemoveCash(target, amount)
exports.nord_inventory:AddDirtyMoney(target, amount, metadata)
exports.nord_inventory:RemoveDirtyMoney(target, amount)
```

Also available: `CanCarryWeight` and `CanSwapItem`.

Do not assume a successful client animation means an item mutation succeeded. Check the export return value and handle failure paths.

Server Exports · Read & Search

Server Exports · Read & Search

Inventory snapshots, slot queries and count/search helpers.

Applies to: `Nord Inventory v2.0.0`

Key exports:

```
GetInventory
GetInventories
GetInventoryItems
GetInventorySnapshot
GetSlot
GetSlotWithItem
GetSlotsWithItem
GetSlotIdWithItem
GetSlotIdsWithItem
GetEmptySlot
GetSlotForItem
GetItemCount
GetItemSlots
GetItem
SearchInventory
Search
```

Use metadata arguments when item identity depends on instance metadata. `strict` matching is useful when an integration requires an exact metadata shape/value match instead of a looser subset match.

Server Exports · Metadata & Inventory Management

Server Exports · Metadata & Inventory Management

Metadata mutation plus capacity and inventory lifecycle controls.

Applies to: `Nord Inventory v2.0.0`

```
SetMetadata(target, slot, metadata)
GetMetadata(target, slot)
UpdateMetadata(target, slot, metadata)
RepairInventoryCapacity(target)
SetMaxWeight(target, maxWeight)
SetSlotCount(target, slots)
ClearInventory(target, keep)
RemoveInventory(target)
```

These are trusted server operations. Avoid exposing them 1:1 through unrestricted net events.

Client Read-Only Exports

Client Read-Only Exports

Local snapshot helpers for UI/gameplay checks.

Applies to: `Nord Inventory v2.0.0`

Current client read exports:

```
Search(searchType, item, metadata)
GetPlayerItems()
GetPlayerWeight()
GetPlayerMaxWeight()
GetItemCount(itemName, metadata, strict)
GetSlotWithItem(itemName, metadata, strict)
GetSlotsWithItem(itemName, metadata, strict)
GetSlotIdWithItem(itemName, metadata, strict)
GetSlotIdsWithItem(itemName, metadata, strict)
OpenInventory(...)
openInventory(...)
```

Client data is ideal for presentation and convenience checks. Final permission/item transfer decisions belong on the server.

Registering Stashes & Custom Inventories

Registering Stashes & Custom Inventories

Dynamic storage registration from another server resource.

Applies to: `Nord Inventory v2.0.0`

```
exports.nord_inventory:RegisterStash(
    'mechanic_storage',
    'Mechanic Storage',
    80,
    250000,
    false,
    { mechanic = 0 },
    vector3(0.0, 0.0, 0.0)
)
```

For more general storage:

```
exports.nord_inventory:RegisterInventory('my_inventory_id', {
    label = 'My Inventory',
    slots = 40,
    maxWeight = 100000
})
```

Register storage from trusted server code and open it through the public inventory API.

Usable Items & External Handlers

Usable Items & External Handlers

Attach behavior without making the browser authoritative.

Applies to: `Nord Inventory v2.0.0`

External item use exports

Nord Inventory has two mutually exclusive usable-item modes:

- `internal`: Nord handles duration, animation, hand prop, player effects and optional removal.
- `export`: Nord validates the item/slot transaction and delegates the gameplay action to another resource export.

The Admin Studio exposes this under **Item Management → Behavior → Use handler** with **Internal (Nord)** and **External export**.

OX-style item definition

A resource can register an item and point its use directly at one of its own exports. When the export name is bare, Nord automatically binds it to the resource calling `RegisterItem`.

```
exports.nord_inventory:RegisterItem('scanner', {
    label = 'Scanner',
    weight = 250,
```

```
    client = {
        export = 'UseScanner', -- automatically <calling_resource>:UseScanner
        remove = 0
    }
})

exports('UseScanner', function(item, slot)
    -- item.name, item.label, item.metadata, item.count, item.slot, ...
    -- Return false to cancel the transaction.
    return true
end)
```

The full resource name is still accepted:

```
client = { export = 'my_scanner:UseScanner', remove = 0 }
-- or
client = { export = 'my_scanner.UseScanner', remove = 0 }
```

Server export

```
exports.nord_inventory:RegisterItem('repair_kit', {
    label = 'Repair Kit',
    weight = 500,
    close = true,
    server = {
        export = 'UseRepairKit',
        remove = 1
    }
})

exports('UseRepairKit', function(source, item, slot, definition)
    -- Return false to cancel and prevent Nord from removing the item.
    return true
end)
```

The Nord-native descriptor is also supported:

```
exports.nord_inventory:RegisterItem('repair_kit', {
    label = 'Repair Kit',
    weight = 500,
    use = {
        mode = 'export',
        export = 'my_mechanic:UseRepairKit',
        side = 'server',
        remove = 1
    }
})
```

`mode = 'export'` may be omitted when `use.export` is present.

Register an external handler for an existing item

Compact form, bound to the calling resource:

```
exports.nord_inventory:RegisterUseExport('repair_kit', 'UseRepairKit', 'server')
```

Explicit resource form:

```
exports.nord_inventory:RegisterUseExport('repair_kit', 'my_mechanic', 'UseRepairKit',
'server')
```

Descriptor/string forms through `RegisterUsableItem` are also accepted:

```
exports.nord_inventory:RegisterUsableItem('repair_kit', 'UseRepairKit', 'server')
exports.nord_inventory:RegisterUsableItem('repair_kit', 'my_mechanic:UseRepairKit', 'server')
exports.nord_inventory:RegisterUsableItem('scanner', { export = 'UseScanner', side = 'client'
})
```

A normal server Lua callback remains supported:

```
exports.nord_inventory:RegisterUsableItem('repair_kit', function(source, item, definition,
slot)
    return true
```

```
end)
```

Register the item and handler in one call

The third `RegisterItem` argument can be a function, export string or export descriptor. The optional fourth argument selects `server` or `client` for string handlers.

```
exports.nord_inventory:RegisterItem('repair_kit', {
    label = 'Repair Kit',
    weight = 500,
    use = { remove = 1 }
}, 'UseRepairKit', 'server')
```

Bulk registration

`RegisterItems` registers multiple runtime items from one resource and applies the same caller-resource binding rules.

```
local ok, result = exports.nord_inventory:RegisterItems({
    scanner = {
        label = 'Scanner',
        weight = 250,
        client = { export = 'UseScanner', remove = 0 }
    },
    repair_kit = {
        label = 'Repair Kit',
        weight = 500,
        server = { export = 'UseRepairKit', remove = 1 }
    }
})
```

Transaction behavior

Nord validates that the player still owns the same item/slot before the external action is committed. If an export is unavailable, errors, or returns `false`, the use is cancelled and Nord does not remove the item.

`remove` stays server-authoritative:

- `remove = 0`: keep the item.
- `remove = 1`: remove one item after a successful external action.
- Higher values remove that amount, limited by the current stack.

External mode is exclusive: Nord does **not** also run its internal duration, animation, hand prop, hunger, thirst, health or armor effects. The external resource owns those effects.

Hands-only animations (`prop = false`)

An internal usable item can explicitly disable every hand-prop fallback while keeping its animation.

```
use = {
    duration = 2500,

    animation = {
        dict = 'missheistdockssetup1clipboard@idle_a',
        clip = 'idle_a',
        flag = 49
    },

    prop = false
}
```

`prop = false` is authoritative for the use action. Nord Inventory will not fall back to:

- `worldModel`;
- keyword prop rules;
- category prop rules;
- the generic fallback prop.

The item's `worldModel` can still be kept for ground/drop presentation.

The default `bandage` uses this mode so the player performs a hands-only bandaging animation with no object attached.

Additional handler exports:

```
RegisterUsableItem
RegisterUseExport
UnregisterUseHandler
```

Developer Ecosystem SDK

Developer Ecosystem SDK

Provider-aware runtime items, actions, schemas, images, hooks and health reporting.

Applies to: `Nord Inventory v2.0.0`

Developer Ecosystem

Nord Inventory exposes a provider-aware server SDK so another resource can own its items and behavior without editing `shared/items.lua` or the Nord core.

Core rules

- Runtime registrations are transient and should be repeated when the provider resource starts.
- The calling FiveM resource is detected with `GetInvokingResource()`; integrations do not need to hard-code their own provider name for bare export handlers.
- A runtime definition cannot replace a database-authoritative item. `RegisterItem` / `RegisterIntegration` return `database_authoritative` for that definition.
- A provider may still attach an external use handler to an already-authoritative item.
- Runtime ownership, use handlers, item actions, metadata schemas and hooks are cleaned up by provider ownership when the resource stops or calls `UnregisterIntegration()`.
- One provider cannot silently steal a runtime item, external use handler, metadata schema or an action ID owned by another provider.

Capability detection

```
if not exports.nord_inventory:HasFeature('integration_manifest') then
    return
end
```

```
local capabilities = exports.nord_inventory:GetCapabilities()
```

Useful capability names include `runtime_items`, `external_use`, `provider_registry`, `integration_manifest`, `item_actions`, `metadata_schema_api`, `durability_api`, `containers`, `image_registration`, `health_check` and `backups`.

RegisterIntegration

Use one server-side manifest when your resource owns several pieces of an integration:

```
local ok, result = exports.nord_inventory:RegisterIntegration({
    name = 'Example Medical',
    version = '1.0.0',
    author = 'Example Studio',
    description = 'Native Nord Inventory integration',
    dependencies = { 'nord_inventory' },

    items = {
        medical_scanner = {
            label = 'Medical Scanner',
            description = 'Portable diagnostic device',
            weight = 1200,
            stack = false,
            close = true,

            -- Optional asset copied from this provider resource.
            imageSource = 'install/nord_inventory/medical_scanner.png',

            client = {
                export = 'UseMedicalScanner',
                remove = 0
            },

            metadataDefaults = {
                serial = '',
                owner = ''
            },
```

```
            metadataSchema = {
                serial = { type = 'text', label = 'Serial' },
                owner = { type = 'text', label = 'Owner' }
            },

            actions = {
                {
                    id = 'inspect',
                    label = 'Inspect Scanner',
                    icon = 'fa-magnifying-glass',
                    side = 'client',
                    export = 'InspectMedicalScanner',
                    close = false
                }
            }
        }
    },

    hooks = {
        itemUsed = function(payload)
            -- Provider-scoped hook; removed automatically with the integration.
        end
    }
})
```

Client exports in the provider resource:

```
exports('UseMedicalScanner', function(item, slot)
    StartMedicalScanner(item, slot)
    return true
end)

exports('InspectMedicalScanner', function(item, slot, actionId)
    InspectScanner(item, slot)
    return true
end)
```

RegisterItem / RegisterItems

```
local ok, def = exports.nord_inventory:RegisterItem('service_token', {
    label = 'Service Token',
    weight = 5,
    stack = true,
    close = false,
    metadataDefaults = { issuer = '', issuedAt = 0 },
    client = { export = 'UseServiceToken', remove = 0 }
})
```

Batch form:

```
exports.nord_inventory:RegisterItems({
    item_one = { label = 'Item One', weight = 100 },
    item_two = { label = 'Item Two', weight = 200 }
})
```

Use handlers

Attach a callback/funcref:

```
exports.nord_inventory:RegisterUsableItem('legacy_item', function(source, item, definition,
slot)
    return true
end, 'server')
```

Or attach an external export:

```
exports.nord_inventory:RegisterUseExport('carplay', 'carplay', 'client')
```

Full explicit form:

```
exports.nord_inventory:RegisterUseExport('service_token', {
    resource = 'my_resource',
    export = 'UseServiceToken',
    side = 'server'
})
```

Cross-resource FiveM funcrefs are accepted and wrapped safely; they are not indexed as ordinary Lua tables.

Unregister ownership

```
exports.nord_inventory:UnregisterItem('service_token')
exports.nord_inventory:UnregisterItems({ 'item_one', 'item_two' })
exports.nord_inventory:UnregisterUseHandler('legacy_item')
exports.nord_inventory:UnregisterIntegration()
```

These calls are provider-scoped. A resource cannot unregister another provider's owned registration.

Provider inspection

```
local provider = exports.nord_inventory:GetRegisteredProvider('service_token')
local providers = exports.nord_inventory:GetProviderRegistry()
local debug = exports.nord_inventory:DebugItem('service_token')
```

`DebugItem` reports the definition source, provider, registered actions, metadata extension and use-handler presence.

Item Actions API

Add an action to an existing item context menu:

```
exports.nord_inventory:RegisterItemAction('id_card', {
    id = 'verify_card',
    label = 'Verify Card',
    icon = 'fa-id-card',
    side = 'server',
    export = 'VerifyCard',
    close = false
})

exports('VerifyCard', function(source, item, slot, definition, actionId)
    -- Server-side validation here.
    return true
end)
```

Remove only your own action:

```
exports.nord_inventory:UnregisterItemAction('id_card', 'verify_card')
```

Read public actions:

```
local actions = exports.nord_inventory:GetItemActions('id_card')
```

Metadata Schema API

```
exports.nord_inventory:RegisterMetadataSchema('employee_card', {
    employeeName = { type = 'text', label = 'Employee' },
    company = { type = 'text', label = 'Company' },
    employeeId = { type = 'text', label = 'Employee ID' }
}, {
    employeeName = true,
    company = true,
    employeeId = true
})
```

The extension is merged into the item metadata presentation without rewriting the authoritative item definition.

```
local schema = exports.nord_inventory:GetMetadataSchema('employee_card')
exports.nord_inventory:UnregisterMetadataSchema('employee_card')
```

Item Image Registration

Trusted server resources can copy a PNG/WebP shipped inside their own resource:

```
local ok, imageName = exports.nord_inventory:RegisterItemImage(
    'install/nord_inventory/service_token.png',
    'service_token.png'
)
```

The API rejects traversal paths, unsupported formats, oversized images and target-name ownership conflicts.

For `RegisterIntegration`, use `imageSource` directly inside an item to copy the asset and assign the resulting image name to the definition.

Durability API

```
local value, maximum = exports.nord_inventory:GetDurability(source, slot)
exports.nord_inventory:SetDurability(source, slot, 50)
exports.nord_inventory:AddDurability(source, slot, 20)
exports.nord_inventory:RemoveDurability(source, slot, 10)
exports.nord_inventory:RepairItem(source, slot)      -- full repair
exports.nord_inventory:RepairItem(source, slot, 25) -- add 25
```

All mutations run through the normal Nord inventory metadata path and sync the affected inventory.

Containers

```
exports.nord_inventory:OpenItemContainer(source, slot)
local container = exports.nord_inventory:GetContainerInventory(source, slot)
```

The slot must contain an item whose Nord definition has a `container` configuration.

Hooks

```
exports.nord_inventory:RegisterHook('itemUsed', function(payload)
    print(payload.source, payload.item and payload.item.name)
end)
```

Provider-owned hooks are removed on `UnregisterIntegration()` or resource stop.

Hook names emitted by the current core include `beforeUseItem`, `itemUsed`, `itemAdded`, `itemRemoved`, `itemMoved`, `metadataChanged`, `itemRepaired`, `itemBroken`, `itemAction`, `inventoryOpened` and `inventoryClosed`.

Health diagnostics

```
local report = exports.nord_inventory:GetHealthReport()
```

The report is also visible in Admin Studio and checks provider/resource state, dependencies, export descriptors, missing images, durability configuration, registration conflicts and orphan extensions.

Metadata privacy from `RegisterItem`

External resources can decide whether an item's metadata detail cards are visible to players directly when registering the item.

Use the canonical option:

```
exports['nord_inventory']:RegisterItem('nr_food', {
    label = 'Restaurant Food',
    weight = 250,
    stack = true,

    -- Player can still see metadata.label / metadata.image as presentation,
    -- but the remaining metadata values are hidden from the item-info modal.
    hideMetadataDetails = true
})
```

Then an instance can still use normal presentation metadata:

```
exports['nord_inventory']:AddItem(source, 'nr_food', 1, {
    label = 'Cute Cafe Burger',
    image = 'cute_cafe_burger.png',

    recipeId = 14,
    restaurant = 'cute_cafe',
    hunger = 25,
    basePrice = 20
})
```

The player sees `Cute Cafe Burger` and `cute_cafe_burger.png`, but does not see `recipeId`, `restaurant`, `hunger`, `basePrice`, or other metadata detail cards.

The metadata is **not deleted or filtered from server exports**. Scripts still receive the complete metadata.

For compatibility, these registration aliases are also accepted:

```
hideMetadata = true
metadataPrivate = true
metadataPlayerVisible = false
metadataVisibility = false
metadataVisibility = 'hidden'
```

Nord normalizes all of them internally to:

```
hideMetadataDetails = true
```

The same option works with `RegisterItems` and with each item inside `RegisterIntegration`.

Metadata details are hidden by default

All registered items now use a privacy-first default.

This is enough:

```
exports['nord_inventory']:RegisterItem('nr_food', {
    label = 'Restaurant Food',
    weight = 250
})
```

Metadata detail cards are hidden automatically.

Instance presentation still works:

```
exports['nord_inventory']:AddItem(source, 'nr_food', 1, {
    label = 'Cute Cafe Burger',
    image = 'cute_cafe_burger.png',
    recipeId = 14,
    restaurant = 'cute_cafe'
})
```

The player sees `Cute Cafe Burger` and its metadata image, but does not see `recipeId`, `restaurant`, or other metadata cards.

To deliberately show metadata details for one item:

```
exports['nord_inventory']:RegisterItem('identity_card', {
    label = 'Identity Card',
    weight = 10,
    hideMetadataDetails = false
})
```

The equivalent opt-in alias is:

```
showMetadataDetails = true
```

You can still use per-field `metadataDisplay` / schema `show` settings once the item has opted into visible metadata details.

Register items together with their category

For integrations such as mechanics, restaurants or jobs, you can register the category and all items in one call. Every item automatically receives the category; you do not need to repeat `category = 'mechanic'` on each definition.

```
local ok, result = exports['nord_inventory']:RegisterCategoryItems({
    name = 'mechanic_materials',
    label = 'Mechanic Materials',
    icon = 'fa-screwdriver-wrench',
    order = 80
}, {
    mech_sandpaper = {
        label = 'Sandpaper',
        weight = 150,
        stack = true
    },

    mech_primer = {
```

```
        label = 'Paint Primer',
        weight = 500,
        stack = true
    },

    mech_clearcoat = {
        label = 'Clear Coat',
        weight = 500,
        stack = true
    }
})
```

The result includes the category state plus registered/failed items:

```
{
    category = 'mechanic_materials',
    categoryState = 'created', -- or existing
    registered = { 'mech_sandpaper', 'mech_primer', 'mech_clearcoat' },
    failed = {}
}
```

The equivalent alias is:

```
exports['nord_inventory']:RegisterItemsWithCategory(category, items)
```

A string category name is also supported with the category definition as the third argument:

```
exports['nord_inventory']:RegisterCategoryItems(
    'mechanic_materials',
    items,
    { label = 'Mechanic Materials', icon = 'fa-screwdriver-wrench' }
)
```

Register a complete item catalog

Use `RegisterItemCatalog` when one resource owns several categories:

```
exports['nord_inventory']:RegisterItemCatalog({
    mechanic_paint = {
        label = 'Paint Materials',
```

```
        icon = 'fa-spray-can-sparkles',
        order = 80,
        items = {
            mech_basecoat = { label = 'Base Coat', weight = 500 },
            mech_clearcoat = { label = 'Clear Coat', weight = 500 }
        }
    },

    mechanic_tools = {
        label = 'Mechanic Tools',
        icon = 'fa-screwdriver-wrench',
        order = 90,
        items = {
            mech_polisher = { label = 'Polisher', weight = 1800, stack = false },
            mech_wrench = { label = 'Workshop Wrench', weight = 900, stack = false }
        }
    }
})
```

Categories are created/reused first, then their items are registered. Runtime items still follow the normal provider ownership rules and should be registered again when the provider resource starts. Custom category folders are persistent.

Dynamic category registration

External resources can install their own item categories before registering items. Category registration is idempotent: if the category already exists, Nord Inventory reuses it instead of creating a duplicate.

Register one category

```
local ok, category, state = exports['nord_inventory']:RegisterCategory(
    'restaurant_food',
    {
        label = 'Restaurant Food',
        icon = 'fa-utensils',
        order = 100
    }
```

```
)
```

`state` is `created` or `existing`.

Register several categories

```
exports['nord_inventory']:RegisterCategories({
    restaurant_food = {
        label = 'Restaurant Food',
        icon = 'fa-utensils',
        order = 100
    },

    restaurant_drinks = {
        label = 'Restaurant Drinks',
        icon = 'fa-glass-water',
        order = 110
    }
})
```

Create a category directly from `RegisterItem`

An item can carry an inline category definition:

```
exports['nord_inventory']:RegisterItem('nr_burger', {
    label = 'Restaurant Burger',
    weight = 250,

    category = {
        name = 'restaurant_food',
        label = 'Restaurant Food',
        icon = 'fa-utensils',
        order = 100
    }
})
```

Nord creates/reuses `restaurant_food` first, then registers the item with:

```
category = 'restaurant_food'
```

`RegisterIntegration` categories

This is the recommended form for complete script integrations:

```
exports['nord_inventory']:RegisterIntegration({
    name = 'Nord Restaurants',

    categories = {
        restaurant_food = {
            label = 'Restaurant Food',
            icon = 'fa-utensils',
            order = 100
        },

        restaurant_drinks = {
            label = 'Restaurant Drinks',
            icon = 'fa-glass-water',
            order = 110
        }
    },

    items = {
        nr_burger = {
            label = 'Burger',
            category = 'restaurant_food',
            weight = 250
        },

        nr_coffee = {
            label = 'Coffee',
            category = 'restaurant_drinks',
            weight = 300
        }
    }
})
```

Categories are registered before items in the same integration call.

Existing categories

If a category already exists, it is reused and is not overwritten by another resource. A resource that originally created a custom category may explicitly update it:

```
exports['nord_inventory']:RegisterCategory('restaurant_food', {
    label = 'Restaurant Meals',
    icon = 'fa-utensils',
    order = 90,
    updateExisting = true
})
```

Only the owning resource can use `updateExisting = true`.

Read categories

```
local category = exports['nord_inventory']:GetCategory('restaurant_food')
local categories = exports['nord_inventory']:GetCategories()
```

Remove an owned category

```
local ok, reason = exports['nord_inventory']:UnregisterCategory('restaurant_food')
```

For safety, a category can only be removed by the resource that created it and only while it contains no active items. `UnregisterIntegration` does not delete persistent category folders automatically.

Categories & Runtime Catalogs

Categories & Runtime Catalogs

Register categories and grouped catalogs from provider resources.

Applies to: `Nord Inventory v2.0.0`

Category exports:

```
RegisterCategory
RegisterCategories
GetCategory
GetCategories
UnregisterCategory
RegisterCategoryItems
RegisterItemsWithCategory
RegisterItemCatalog
```

Grouped registration is preferred for resources such as restaurants/crafting where many related items share one category/provider identity.

Integration Manifests & Provider Ownership

Integration Manifests & Provider Ownership

Register a resource as an explicit provider and expose diagnostics.

Applies to: `Nord Inventory v2.0.0`

```
exports.nord_inventory:RegisterIntegration({
    name = 'My Resource',
    version = '1.0.0',
    author = 'My Studio',
    description = 'Adds custom items and actions.'
})
```

Unregister with:

```
exports.nord_inventory:UnregisterIntegration()
```

Provider ownership helps Admin Studio explain which resource owns a runtime definition/action/image and whether that provider is currently online.

Item Actions, Metadata Schemas & Images

Item Actions, Metadata Schemas & Images

Presentation and action extension points.

Applies to: `Nord Inventory v2.0.0`

```
RegisterItemAction(itemName, action)
UnregisterItemAction(itemName, actionId)
RegisterMetadataSchema(itemName, schema, display)
UnregisterMetadataSchema(itemName)
RegisterItemImage(sourcePath, targetName)
GetItemActions(itemName)
GetMetadataSchema(itemName)
```

A schema describes expected/display metadata; it does not make client-provided metadata trustworthy. Validate business rules on the server.

Durability & Container API

Durability & Container API

Lifecycle and nested-storage helpers for integrations.

Applies to: `Nord Inventory v2.0.0`

Durability:

```
GetDurability(target, slot)
SetDurability(target, slot, value)
AddDurability(target, slot, amount)
RemoveDurability(target, slot, amount)
RepairItem(target, slot, amount)
```

Containers:

```
OpenItemContainer(source, slot)
GetContainerInventory(target, slot)
```

Respect server-defined capacity/depth and item lifecycle rules rather than duplicating them in another resource.

Hooks, Capabilities & Diagnostics

Hooks, Capabilities & Diagnostics

Feature detection and operational inspection.

Applies to: `Nord Inventory v2.0.0`

```
RegisterHook(name, callback)
registerHook(name, callback)
HasFeature(name)
GetCapabilities()
GetRegisteredProvider(itemName)
GetProviderRegistry()
GetHealthReport()
GetIntegrationInfo()
DebugItem(itemName)
```

Feature detection is safer than version-string branching when you only need to know whether a capability is present.

Admin Access Exports

Admin Access Exports

Public server access bridge used by Nord Staff and trusted administration tools.

Applies to: `Nord Inventory v2.0.0`

```
HasAdminAccess(target, maybeTarget)
GiveAdminAccess(target, maybeTarget)
RemoveAdminAccess(target, maybeTarget)
```

Runtime grants can persist in `nord_inventory_admin_access`. They do not need to rewrite ACE or static license configuration.

Player Interaction Exports

Player Interaction Exports

Open search/robbery sessions and integrate surrender state.

Applies to: `Nord Inventory v2.0.0`

Server:

```
OpenPlayerSearch(source, target)
OpenPlayerRobbery(source, target)
IsPlayerSurrendered(target)
```

Client:

```
SetHandsUpState(value)
IsHandsUp()
```

Use the server exports for authority-sensitive actions.

Notify & Text UI Exports

Notify & Text UI Exports

Use Nord bridges without binding your integration to the NUI implementation.

Applies to: `Nord Inventory v2.0.0`

Server notify:

```
exports.nord_inventory:Notify(target, message, kind, duration, title)
```

Server TextUI:

```
ShowTextUI(target, text, options)
HideTextUI(target)
ShowTextUIEntry(target, id, data)
HideTextUIEntry(target, id)
```

Bridge provider selection is controlled in `config/bridges.lua`.

OX / QB Migration Examples

OX / QB Migration Examples

Common compatibility patterns when moving an integration to Nord Inventory.

Applies to: `Nord Inventory v2.0.0`

Count

```
local count = exports.nord_inventory:GetItemCount(source, 'water')
```

Add/remove

```
if exports.nord_inventory:CanCarryItem(source, 'water', 2) then
    exports.nord_inventory:AddItem(source, 'water', 2)
end
exports.nord_inventory:RemoveItem(source, 'water', 1)
```

Search a slot

```
local slot = exports.nord_inventory:GetSlotWithItem(source, 'water')
```

Canonicalize a legacy name

```
local item = exports.nord_inventory:ResolveItemName(legacyName)
```

Migrate behavior at the public export boundary. Avoid emulating another inventory's private events/database tables inside Nord.

Complete Export Index

Complete Export Index

Generated from export declarations in this exact build.

Applies to: `Nord Inventory v2.0.0`

Server · Inventory & SDK

- `GetInventoryType()`
- `GetItemDefinition(name)`
- `GetItems()`
- `GetItemList()`
- `ResolveItemName(value)`
- `GetExternalItemDefinition(value)`
- `GetItemCatalog()`
- `GetItemImage(value, mode)`
- `GetItemImageData(value)`
- `RegisterCategory(name, data)`
- `RegisterCategories(definitions)`
- `GetCategory(name)`
- `GetCategories()`
- `UnregisterCategory(name)`
- `RegisterItem(name, definition, useHandler, side)`
- `RegisterItems(definitions)`
- `RegisterInventory(id, options)`
- `Items(itemName)`
- `ItemList(itemName)`
- `OpenInventory()`
- `openInventory()`
- `forceOpenInventory()`
- `RegisterStash(id, label, slots, maxWeight, owner, groups, coords)`
- `AddItem(target, itemName, amount, metadata, slot, cb)`
- `AddCash(target, amount, metadata)`
- `RemoveCash(target, amount)`
- `AddDirtyMoney(target, amount, metadata)`

- `RemoveDirtyMoney(target, amount)`
- `RemoveItem(target, itemName, amount, metadata, slot, ignoreTotal, strict)`
- `CanCarryItem(target, itemName, amount, metadata)`
- `CanCarryWeight(target, weight)`
- `CanCarryAmount(target, itemName)`
- `CanSwapItem(target, firstItem, firstItemCount, testItem, testItemCount)`
- `GetInventory(target, owner)`
- `GetInventories(invType, detailed)`
- `GetInventoryItems(target, owner)`
- `GetInventorySnapshot(target, owner)`
- `GetIntegrationInfo()`
- `GetSlot(target, slot)`
- `GetSlotWithItem(target, itemName, metadata, strict)`
- `GetSlotsWithItem(target, itemName, metadata, strict)`
- `GetSlotIdWithItem(target, itemName, metadata, strict)`
- `GetSlotIdsWithItem(target, itemName, metadata, strict)`
- `GetEmptySlot(target)`
- `GetSlotForItem(target, itemName, metadata)`
- `GetItemCount(target, itemName, metadata, strict)`
- `GetItemSlots(target, itemName, metadata)`
- `GetItem(target, itemName, metadata, returnsCount)`
- `SearchInventory(target, searchType, itemName, metadata)`
- `Search(target, searchType, itemName, metadata)`
- `SetMetadata(target, slot, metadata)`
- `GetMetadata(target, slot)`
- `UpdateMetadata(target, slot, metadata)`
- `RepairInventoryCapacity(target)`
- `SetMaxWeight(target, maxWeight)`
- `SetSlotCount(target, slots)`
- `ClearInventory(target, keep)`
- `RemoveInventory(target)`
- `RegisterUsableItem(itemName, handler, side)`
- `RegisterUseExport(itemName, resourceOrExport, exportNameOrSide, side)`
- `UnregisterUseHandler(itemName)`
- `RegisterHook(name, callback)`
- `registerHook(name, callback)`
- `HasFeature(name)`
- `GetCapabilities()`
- `GetRegisteredProvider(itemName)`
- `RegisterItemAction(itemName, action)`
- `UnregisterItemAction(itemName, actionId)`
- `RegisterItemImage(sourcePath, targetName)`
- `UnregisterItem(itemName)`
- `UnregisterItems(items)`
- `RegisterCategoryItems(category, items, categoryData)`

- `RegisterItemsWithCategory(category, items, categoryData)`
- `RegisterItemCatalog(catalog)`
- `RegisterIntegration(manifest)`
- `UnregisterIntegration()`
- `RegisterMetadataSchema(itemName, schema, display)`
- `UnregisterMetadataSchema(itemName)`
- `OpenItemContainer(source, slot)`
- `GetContainerInventory(target, slot)`
- `DebugItem(itemName)`
- `GetDurability(target, slot)`
- `SetDurability(target, slot, value)`
- `AddDurability(target, slot, amount)`
- `RemoveDurability(target, slot, amount)`
- `RepairItem(target, slot, amount)`
- `GetItemActions(itemName)`
- `GetMetadataSchema(itemName)`
- `GetProviderRegistry()`
- `GetHealthReport()`

Client · Read-only inventory

- `Search(target, searchType, itemName, metadata)`
- `GetPlayerItems()`
- `GetPlayerWeight()`
- `GetPlayerMaxWeight()`
- `GetItemCount(target, itemName, metadata, strict)`
- `GetSlotWithItem(target, itemName, metadata, strict)`
- `GetSlotsWithItem(target, itemName, metadata, strict)`
- `GetSlotIdWithItem(target, itemName, metadata, strict)`
- `GetSlotIdsWithItem(target, itemName, metadata, strict)`

Client · Runtime

- `OpenInventory()`
- `openInventory()`

Server · Notify

- `Notify(target, message, kind, duration, title)`

Server · Text UI

- `ShowTextUI(target, text, options)`
- `HideTextUI(target)`
- `ShowTextUIEntry(target, id, data)`
- `HideTextUIEntry(target, id)`

Server · Player interactions

- `OpenPlayerSearch(source, target)`
- `OpenPlayerRobbery(source, target)`
- `IsPlayerSurrendered(target)`

Client · Player interactions

- `SetHandsUpState(value)`
- `IsHandsUp()`

Server · Vehicle storage

- `GetVehicleStorageCapacity(invType, modelHash, vehicleClass)`
- `GetVehicleTrunkWeight(modelHash, vehicleClass)`
- `GetVehicleTrunkSlots(modelHash, vehicleClass)`
- `GetVehicleGloveboxWeight(modelHash, vehicleClass)`
- `GetVehicleGloveboxSlots(modelHash, vehicleClass)`

Server · Admin access

- `HasAdminAccess(target, maybeTarget)`
- `GiveAdminAccess(target, maybeTarget)`
- `RemoveAdminAccess(target, maybeTarget)`

Server · Operations / Forensics

- `GetItemPolicy(itemName)`
- `CreateInventorySnapshot(invKey, reason)`
- `GetInventoryLedger(filters)`
- `FindItemInstances(filters)`

Operations API · v2.0.0

Operations API · v2.0.0

Applies to: `Nord Inventory v2.0.0`

Four server exports are added by the Operations module. They are intended for trusted server resources/tooling.

GetItemPolicy

```
local policy = exports.nord_inventory:GetItemPolicy('police_evidence')
```

Returns the resolved movement-policy data for the canonical item name.

CreateInventorySnapshot

```
local ok, snapshotId, canonicalKey = exports.nord_inventory:CreateInventorySnapshot(invKey,
'manual_tool')
```

Creates a snapshot using live runtime state when the target player is connected and persisted state when offline. The invoking resource is recorded as the creator.

GetInventoryLedger

```
local rows = exports.nord_inventory:GetInventoryLedger({
    search = 'weapon_pistol',
    limit = 50
})
```

Returns bounded transaction/forensics results. Treat this as a read/query API, not a substitute for inventory mutations.

FindItemInstances

```
local rows = exports.nord_inventory:FindItemInstances({
    item = 'weapon_pistol',
    serial = 'SERIAL-HERE'
})
```

Searches persisted and live cached inventory state using the same overlay rules as Operations so connected inventories are not double-counted from stale SQL.

Security

Do not expose these exports 1:1 through unrestricted client net events. Snapshot creation and forensic data should remain server-side/trusted. Admin Studio actions still use the normal permission bridge.