

Permission Resource

Check APIs and Client Usage

Version 1.2.4-SNAPSHOT, 2026-01-31T15:41:23Z

Table of Contents

1. APIs	2
1.1. POST /permission/check	2
1.2. POST /permission/check-with-index	2
1.3. Response example (legacy rules present and requiresServer=true)	4
2. Data-Domain Scope and Fallback	8
3. JavaScript Client Library	9
4. Single-resource check API: evalMode and SCOPED decisions	10
5. Evaluate API: per-action decisions with SCOPED semantics	12
5.1. Which rule produced the decision?	13
6. Behavior matrix (summary)	14
7. Migration notes	15
8. Troubleshooting / FAQs	16
8.1. Using in a Browser (no build tools)	16
8.2. Using in React	16
8.3. Using in Vue	17
9. cURL Examples	18
10. Caching Guidance	19
11. Troubleshooting	20

This guide explains how to use the Permission Resource check APIs and how to evaluate the server-produced access decisions on the client using the provided JavaScript library.

It covers:

- The `/check` API (server-evaluated permission)
- The `/check-with-index` API (client-evaluatable snapshot)
- Request payloads and detailed response structures
- Scope and data-domain fallback behavior
- Using the JavaScript client in a browser, React, or Vue



This document reflects the current and evolving server behavior. The JavaScript client library is available at runtime from Quarkus as `/security/acl-client.js` and is forward-compatible with the scoped access-matrix format described here.

Chapter 1. APIs

1.1. POST /permission/check

Performs a server-side permission check for a single request using the caller identity and an optional data domain (organization/account/tenant/segment/owner).

Request

```
{  
    "identity": "user-123",  
    "realm": "b2bi",  
    "area": "security",  
    "functionalDomain": "userProfile",  
    "action": "view",  
    "resourceId": "12345",  
    "orgRefName": "acme",  
    "accountNumber": "A1",  
    "tenantId": "t-001",  
    "dataSegment": 0,  
    "ownerId": "user-123",  
    "roles": ["user", "admin"],  
    "scope": "api"  
}
```

Response (example)

```
{  
    "finalEffect": "ALLOW",  
    "winningRule": "SysAnyActionSecurity",  
    "explanations": [  
        { "rule": "SysAnyActionSecurity", "effect": "ALLOW" }  
    ]  
}
```

Notes: - This endpoint is authoritative and understands scripts/postconditions and any dynamic runtime evaluation. - Use `/check` when you must account for scripts, context-enriched rules, or when you don't have the precomputed snapshot.

1.2. POST /permission/check-with-index

Returns a precomputed permission snapshot for the supplied identity. This snapshot can be cached on the client and used for fast allow/deny decisions without server roundtrips.

Depending on server version/config, the response can include:

- A legacy flat list of rules (for backward compatibility)
- A scoped access matrix (recommended) keyed by data-domain scope → area → domain → action

Request (current servers; send data-domain as top-level fields)

```
{  
  "identity": "user-123",  
  "realm": "b2bi",  
  "orgRefName": "acme",  
  "accountNumber": "A1",  
  "tenantId": "t-001",  
  "dataSegment": 0,  
  "ownerId": "user-123"  
}
```



If your server version supports a nested dataDomain object, you may also send:

```
{  
  "identity": "user-123",  
  "realm": "b2bi",  
  "dataDomain": {  
    "orgRefName": "acme",  
    "accountNumber": "A1",  
    "tenantId": "t-001",  
    "dataSegment": 0,  
    "ownerId": "user-123"  
  }  
}
```

On servers that do not support nested dataDomain, this shape will produce: **400 Bad Request: Unrecognized field "dataDomain".**

Response: Scoped access matrix (recommended shape)

```
{  
  "enabled": true,  
  "version": 42,  
  "policyVersion": 123,  
  "sources": ["user:user-123", "role:user", "role:admin"],  
  "requiresServer": false,  
  
  "scopes": {  
    "org=acme|acct=A1|tenant=t-001|seg=0|owner=user-123": {  
      "requiresServer": false,  
      "matrix": {  
        "security": {  
          "userProfile": {  
            "view": { "effect": "ALLOW", "rule": "ViewOwnProfile", "priority": 5,  
"finalRule": true, "source": "role:user" }  
          },  
          "credential": {  
            "update": { "effect": "DENY", "rule": "NoUpdate", "priority": 10,  
"finalRule": true, "source": "role:user" }  
          }  
        }  
      }  
    }  
  }  
}
```

```

    "finalRule": true, "source": "role:user" }
        }
    },
    "orders": {
        "manage": { "*": { "effect": "DENY", "rule": "NoManage", "priority": 50,
"finalRule": true } }
    }
},
{
    "org=acme|acct=A1|tenant=t-001|seg=*|owner=*": { "requiresServer": false, "matrix
": { "*": { "*": { "*": { "effect": "DENY", "rule": "DefaultDeny", "priority": 999,
"finalRule": false } } } } },
    "org=*|acct=*|tenant=*|seg=*|owner=*": { "requiresServer": false, "matrix": {
"security": { "*": { "*": { "effect": "ALLOW", "rule": "SysRoleAnyActionSecurity",
"priority": 1, "finalRule": true } } } }
},
    "requestedScope": "org=acme|acct=A1|tenant=t-001|seg=0|owner=user-123",
"requestedFallback": [
    "org=acme|acct=A1|tenant=t-001|seg=0|owner=*",
    "org=acme|acct=A1|tenant=t-001|seg=*|owner=*",
    "org=acme|acct=A1|tenant=*|seg=*|owner=*",
    "org=acme|acct=*|tenant=*|seg=*|owner=*",
    "org=*|acct=*|tenant=*|seg=*|owner=*"
],
"rules": [
    { "name": "SysRoleAnyActionSecurity", "uri": "system:security:*:*:*:*:*:*",
"effect": "ALLOW", "priority": 1, "finalRule": true }
]
}
}

```

Interpretation: - The client should prefer the scope that best matches its current data domain and then look up `area → domain → action` in that scope's matrix, falling back through `requestedFallback`. - Within a matrix, exact values beat wildcards; the matrix already encodes the winning outcome per triple. - If `requiresServer` is true (globally or for a specific scope), the client should call `/check` for decisions in those affected areas.

1.3. Response example (legacy rules present and `requiresServer=true`)

```
{
    "enabled": false,
    "version": 0,
    "policyVersion": 163044986023000,
    "rules": [
        {
            "name": "users can't delete anything in security area",
            "uri": "user:security*:delete|*:*:*:*:*:*",

```

```

    "effect": "DENY",
    "priority": 10,
    "finalRule": true
},
{
  "name": "view your own resources",
  "uri": "user:*:*:system-com:*:*:*:*:*",
  "effect": "ALLOW",
  "priority": 10,
  "finalRule": true
},
{
  "name": "view your own resources, limit to default dataSegment",
  "uri": "user:*:*|*:*:system@system.com:*",
  "effect": "ALLOW",
  "priority": 10,
  "finalRule": false
},
{
  "name": "ViewSystemResources",
  "uri": "user:*:view|system-com:*:*:system@system.com:*",
  "effect": "ALLOW",
  "priority": 10,
  "finalRule": true
}
],
"sources": [
  "user"
],
"requiresServer": true,
"scopes": {
  "org=*|acct=*|tenant=*|seg=*|owner=system@system.com": {
    "matrix": {
      "*": {
        "*": {
          "view": {
            "effect": "ALLOW",
            "rule": "ViewSystemResources",
            "priority": 10,
            "finalRule": true,
            "source": "user"
          }
        }
      }
    }
  },
  "requiresServer": false
},
"org=*|acct=*|tenant=*|seg=*|owner=*": {
  "matrix": {
    "security": {
      "*": {

```

```

        "delete": {
            "effect": "DENY",
            "rule": "users can't delete anything in security area",
            "priority": 10,
            "finalRule": true,
            "source": "user"
        }
    }
},
"*": {
    "*": {
        "*": {
            "effect": "ALLOW",
            "rule": "view your own resources, limit to default dataSegment",
            "priority": 10,
            "finalRule": false,
            "source": "user"
        }
    }
},
"requiresServer": false
}
},
"requestedScope": "org=acme|acct=A1|tenant=t-001|seg=0|owner=user-123",
"requestedFallback": [
    "org=acme|acct=A1|tenant=t-001|seg=0|owner=*",
    "org=acme|acct=A1|tenant=t-001|seg=*|owner=*",
    "org=acme|acct=A1|tenant=*|seg=*|owner=*",
    "org=acme|acct=*|tenant=*|seg=*|owner=*",
    "org=*|acct=*|tenant=*|seg=*|owner=*""
]
}
}

```

Field semantics

- enabled: false indicates the compiled index is disabled or unavailable. Clients should treat the snapshot as non-authoritative and prefer calling /permission/check for critical decisions; the matrix may still be present for some scopes but is not guaranteed complete.
- version: 0 accompanies enabled=false. When enabled is true, version corresponds to the compiled index version and can be used for caching together with policyVersion.
- policyVersion: the ruleset/policy timestamp or version for cache invalidation.
- rules: legacy flat list preserved for backward compatibility. Clients should prefer the scoped matrix when available.
- sources: identities included when the snapshot was materialized (e.g., user id and/or roles).
- requiresServer (top-level): true means at least one rule could not be safely materialized (e.g., uses scripts/postconditions or dynamic filters). Clients should be prepared to call /permission/check for affected scopes/decisions.

- `scopes[<key>].requiresServer`: per-scope flag. If true, client should not rely on that scope's matrix for final decisions and should call `/permission/check` when evaluating in that scope.
- `requestedScope` / `requestedFallback`: convenience keys provided when the request included data-domain values. Clients should attempt lookup starting at `requestedScope`, then walk `requestedFallback` in order.

Chapter 2. Data-Domain Scope and Fallback

A scope key is a canonical string combining the data-domain dimensions:

```
org=<v>|acct=<v>|tenant=<v>|seg=<v>|owner=<v>
```

- Values are specific strings or `*` for wildcard.
- Fallback traversal order: owner → segment → tenant → account → org → global.

Chapter 3. JavaScript Client Library

The JavaScript client provides helpers to evaluate the snapshot on the client.

- Served by Quarkus at: [/security/acl-client.js](#)
- Path in repo: [quantum-framework/src/main/resources/META-INF/resources/security/acl-client.js](#)

Exported API

```
ACLClient.scopeKeyFromDataDomain(dataDomain) // => scope key string
ACLClient.buildFallbackChain(scopeKey)        // => [less-specific scope keys]
ACLClient.lookupAreaDomainAction(matrix, area, domain, action) // => Outcome | null
ACLClient.decide(snapshot, dataDomain, area, domain, action)   // => 'ALLOW' | 'DENY'
ACLClient.decideOutcome(snapshot, dataDomain, area, domain, action) // => Outcome | null
ACLClient.interpretCheckResponse(check) // => { decision, scope, constraints,
filterConstraintsPresent, filterConstraints }
ACLClient.interpretEvaluateResponse(res) // => { allow, deny, decisions, evalModelUsed,
getDecision(area, domain, action) }
```

Outcome structure:

```
{
  "effect": "ALLOW",
  "rule": "<winning rule name>",
  "priority": 0,
  "finalRule": true,
  "source": "role:user"
}
```

Allowed effect values are "ALLOW" or "DENY".

Chapter 4. Single-resource check API: evalMode and SCOPED decisions

The `/system/permissions/check` endpoint accepts an optional `evalMode` parameter to control how the server evaluates filter strings and postcondition scripts in single-resource checks. It also supports sending a shallow snapshot of a concrete resource instance so that filters can be evaluated in-memory when appropriate.

Eval modes:

- LEGACY (default):
 - If a concrete resource instance is provided (with `modelClass` and `resource`), the server attempts in-memory evaluation of rule filter strings; if filters match and postcondition is true/absent, the decision is EXACT.
 - If no resource is provided and a matching rule carries filters and/or scripts, the server returns a SCOPED decision: the outcome (ALLOW/DENY) is accompanied by `scopedConstraints` enumerating the applied filters and/or scripts.
 - List endpoints should continue to use `getFilters(...)/check-with-index` for DB-side filtering; SCOPED simply surfaces constraints to the client for transparency.
- AUTO: Same behavior as LEGACY; provided for client clarity.
- STRICT:
 - With a resource present, the evaluator runs. If filter evaluation returns false, the rule is treated as NOT_APPLICABLE.
 - Without a resource for non-LIST actions, matching rules with filters/scripts produce a SCOPED candidate (constraints listed) instead of being silently allowed/denied. Postconditions are not executed in-memory in this case.
 - LIST is never suppressed; constraints are surfaced.

Request fields (additive):

- `identity` (string, required)
- `realm` (string, optional)
- `area`, `functionalDomain`, `action`, `resourceId` (optional; may be wildcard * unless doing a specific check)
- `modelClass` (string, optional): fully qualified class name or resolvable entity name
- `resource` (object, optional): shallow JSON snapshot of the domain resource
- `evalMode` (string, optional): LEGACY | AUTO | STRICT (defaults to LEGACY)

Response fields (additive and backward compatible):

- `decision` — canonical decision string, mirrors `finalEffect` ("ALLOW" | "DENY").
- `decisionScope` — one of:

- **EXACT** — fully evaluated in-memory (filters and scripts, when present).
- **SCOPED** — conditionally applies; **scopedConstraints** lists the filters/scripts that must hold.
- **DEFAULT** — no rule applied; the decision fell back to the default (**naLabel** will be **NA-ALLOW** or **NA-DENY**).
- **evalModeUsed** — echoes the server-applied eval mode.
- **scopedConstraintsPresent** and **scopedConstraints[]** — present when **decisionScope=SCOPED**.
- **filterConstraintsPresent** and **filterConstraints[]** — legacy-compatible constraint listing; kept for LIST and transition.
- **notApplicable[]** — rules that were considered but did not apply (with phase and reason).
 - Winning rule metadata (additive):
- **winningRuleName**, **winningRulePriority**, **winningRuleFinal** — the rule that produced the decision (for **EXACT**), or the selected candidate (for **SCOPED**). These are null for **DEFAULT**.

Example request enabling evaluator:

```
POST /system/permissions/check
{
  "identity": "alice@end2endlogic.com",
  "realm": "b2bi",
  "area": "sales",
  "functionalDomain": "order",
  "action": "update",
  "modelClass": "com.example.domain.Order",
  "resource": { "id": "ORD-123", "customerId": "5f1e1a5e5e5e5e5e5e5e51" },
  "evalMode": "STRICT"
}
```

Client usage to interpret the decision:

```
const res = await fetch('/system/permissions/check', { method: 'POST', headers: {
  'Content-Type': 'application/json' }, body: JSON.stringify(req) });
const check = await res.json();
const out = ACLClient.interpretCheckResponse(check);
if (out.scope === 'EXACT' && out.decision === 'ALLOW') {
  // fully allowed
} else if (out.scope === 'SCOPED' && out.decision === 'ALLOW') {
  // allowed subject to the following constraints (filters/scripts):
  console.log(out.constraints);
} else {
  // denied or default; inspect check.naLabel / check.notApplicable as needed
}
```

Chapter 5. Evaluate API: per-action decisions with SCOPED semantics

The `/system/permissions/fd/evaluate` endpoint classifies an identity's permissions across discovered areas/domains/actions and now supports the same `evalMode` and optional `modelClass/resource` parameters as `/check`. When the server cannot fully evaluate constraints for a given action (e.g., no resource supplied), it returns a `SCOPED` decision for that action along with the enumerated constraints.

Request (additive parity with `/check`):

```
POST /system/permissions/fd/evaluate?useIndex=true
{
  "identity": "alice@end2endlogic.com",
  "realm": "b2bi",
  // Optional narrowing
  "area": "sales",
  "functionalDomain": "order",
  "action": "view",
  // Optional single-resource evaluation controls
  "modelClass": "com.example.domain.Order",
  "resource": { "id": "ORD-123", "customerId": "5f1e1a5e5e5e5e5e5e5e51" },
  "evalMode": "STRICT"
}
```

Response (additive):

```
{
  "allow": { ... },
  "deny": { ... },
  "decisions": {
    "sales": {
      "order": {
        "view": {
          "effect": "ALLOW",
          "decisionScope": "SCOPED",
          "scopedConstraintsPresent": true,
          "scopedConstraints": [
            { "type": "FILTER", "detail": "customerId:[${accessibleCustomerIds}]" }
          ]
        },
        "naLabel": null,
        "rule": "user-sales-order-view-filter",
        "priority": 100,
        "finalRule": false,
        "source": null
      }
    }
  }
}
```

```
        }
    }
},
"evalModeUsed": "STRICT"
}
```

Client usage:

```
const res = await fetch('/system/permissions/fd/evaluate?useIndex=true', { method:
'POST', headers: { 'Content-Type': 'application/json' }, body: JSON.stringify(req) });
const body = await res.json();
const ev = ACLClient.interpretEvaluateResponse(body);
const decision = ev.getDecision('sales', 'order', 'view');
if (decision && decision.decisionScope === 'EXACT' && decision.effect === 'ALLOW') {
    // fully allowed
} else if (decision && decision.decisionScope === 'SCOPED' && decision.effect ===
'ALLOW') {
    // allowed subject to constraints
    console.log(decision.scopedConstraints);
}
```

5.1. Which rule produced the decision?

For actions resolved by the optimized index, `decisions[area][domain][action]` includes `rule`, `priority`, `finalRule`, and `source` from the index outcome. For server-evaluated fallbacks (when index is disabled or does not cover the specific combination), the server maps the winning rule metadata from the underlying check result, so `rule/priority/finalRule` are also present for parity.

Chapter 6. Behavior matrix (summary)

- With resource present (any mode): evaluator runs. Filters=true and postcondition=true/absent ⇒ **EXACT**. Filters=false ⇒ rule is **NOT_APPLICABLE**. Failures to evaluate ⇒ fallback to SCOPED in STRICT, legacy-compatible otherwise.
- Without resource:
 - **LEGACY/AUTO**: matching rules with filters/scripts surface as **SCOPED** candidates; constraints listed.
 - **STRICT** (non-LIST): also returns **SCOPED**; postconditions are skipped; constraints listed.
 - LIST is never suppressed; DB-side filters continue via `getFilters(…)`; constraints surfaced for transparency.

Chapter 7. Migration notes

- Prefer `evalMode` over the deprecated `enableFilterEval`. If `enableFilterEval=true` and both `modelClass` and `resource` are supplied, the server treats it as `AUTO` for backward compatibility.
- Existing clients that only use `finalEffect` remain compatible. To opt in to richer semantics, read `decision`, `decisionScope`, and `scopedConstraints`.
- For list/search pages, continue to rely on DB-side `getFilters(...)` or `check-with-index`; do not pass a resource snapshot to `/check` for lists.

Chapter 8. Troubleshooting / FAQs

Q: Why do I get **SCOPED** instead of **EXACT**? * A: The server did not have a concrete resource to evaluate, or the predicate engine was not available. Use **STRICT** with **modelClass + resource** for a fully evaluated decision when applicable.

Q: What is **NA-ALLOW / NA-DENY**? * A: They are labels for default fallbacks when no rule could make a decision (**decisionScope=DEFAULT**). The value mirrors the **finalEffect** chosen as the default.

Q: How do I know which rule produced the result? * A: Inspect **winningRuleName**, **winningRulePriority**, **winningRuleFinal** on **/check** responses, or per-action **rule/priority/finalRule** in **/fd/evaluate** responses.

8.1. Using in a Browser (no build tools)

HTML

```
<script src="/security/acl-client.js"></script>
<script>
  async function canViewProfile() {
    // 1) Get a snapshot for the user
    const res = await fetch('/permission/check-with-index', {
      method: 'POST', headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ identity: 'user-123', realm: 'b2bi', orgRefName: 'acme',
accountNumber: 'A1', tenantId: 't-001', dataSegment: 0, ownerId: 'user-123' })
    });
    const snapshot = await res.json();

    // 2) Evaluate locally
    const decision = ACLClient.decide(snapshot, { orgRefName: 'acme', accountNumber:
'A1', tenantId: 't-001', dataSegment: 0, ownerId: 'user-123' }, 'security',
'userProfile', 'view');
    if (decision === 'ALLOW') {
      // show UI
    } else {
      // hide or show alternative
    }
  }
</script>
```

8.2. Using in React

Installation (served by your Quarkus backend)

- No npm package is required; include as an external script in **public/index.html** or via dynamic import.

React example

```
import { useEffect, useState } from 'react';

export default function ProfileButton() {
  const [allowed, setAllowed] = useState(false);

  useEffect(() => {
    async function run() {
      const res = await fetch('/permission/check-with-index', {
        method: 'POST', headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ identity: 'user-123', realm: 'b2bi' })
      });
      const snapshot = await res.json();
      const effect = window.ACClient.decide(snapshot, null, 'security',
        'userProfile', 'view');
      setAllowed(effect === 'ALLOW');
    }
    run();
  }, []);
}

if (!allowed) return null;
return <button>View Profile</button>;
}
```

8.3. Using in Vue

Vue component example

```
<template>
  <button v-if="allowed">View Profile</button>
</template>

<script>
export default {
  data() { return { allowed: false }; },
  async mounted() {
    const res = await fetch('/permission/check-with-index', {
      method: 'POST', headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ identity: 'user-123', realm: 'b2bi' })
    });
    const snapshot = await res.json();
    const effect = window.ACClient.decide(snapshot, null, 'security', 'userProfile',
      'view');
    this.allowed = (effect === 'ALLOW');
  }
}
</script>
```

Chapter 9. cURL Examples

Server-evaluated check

```
curl -sS -X POST \
-H 'Content-Type: application/json' \
http://localhost:8080/permission/check \
-d '{
  "identity": "user-123",
  "realm": "b2bi",
  "area": "security",
  "functionalDomain": "userProfile",
  "action": "view",
  "orgRefName": "acme",
  "accountNumber": "A1",
  "tenantId": "t-001",
  "dataSegment": 0,
  "ownerId": "user-123"
}'
```

Client-evaluatable snapshot

```
curl -sS -X POST \
-H 'Content-Type: application/json' \
http://localhost:8080/permission/check-with-index \
-d '{
  "identity": "user-123",
  "realm": "b2bi",
  "orgRefName": "acme",
  "accountNumber": "A1",
  "tenantId": "t-001",
  "dataSegment": 0,
  "ownerId": "user-123"
}'
```

Chapter 10. Caching Guidance

- Clients should cache the `/check-with-index` response keyed by (`identity`, `realm`, `version`, `policyVersion`).
- Refresh the snapshot when either `version` or `policyVersion` changes.

Chapter 11. Troubleshooting

- If `requiresServer` is `true` (globally or per-scope), call `/check` for affected decisions.
- If no matrix entry is found in any scope fallback, default to `DENY` for safety.
- Ensure `effect` comparisons are case-insensitive on the client (`String(effect).toUpperCase()`).