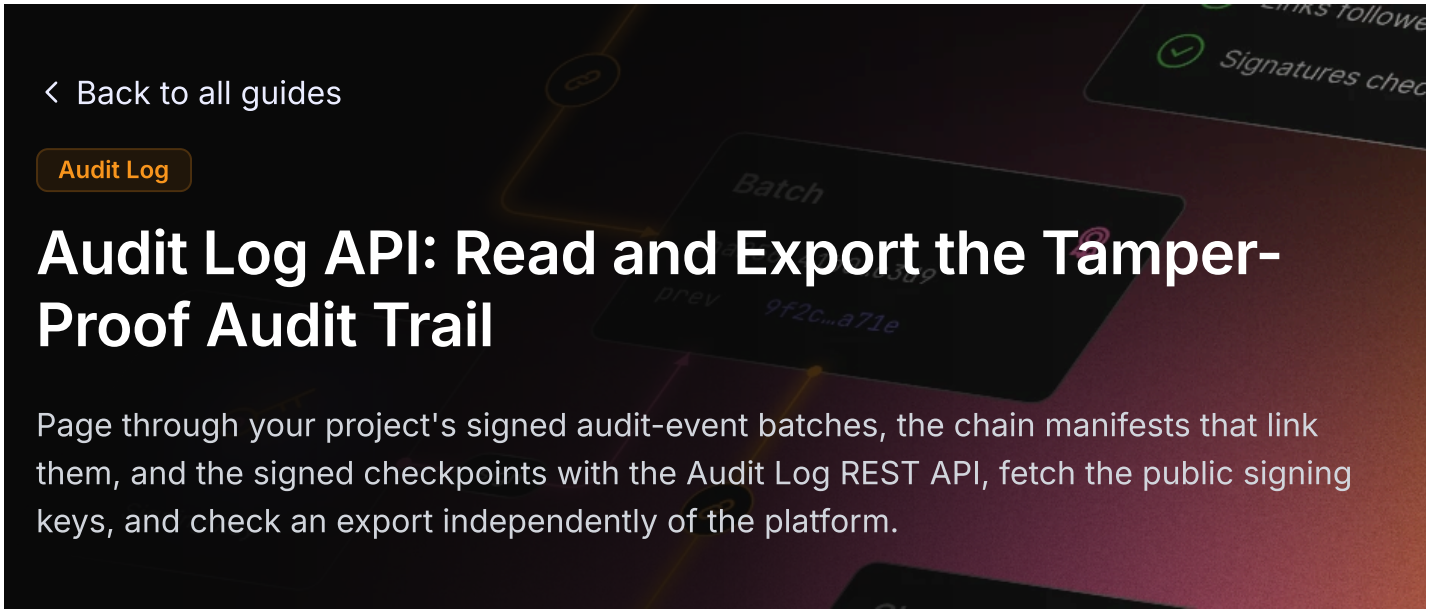
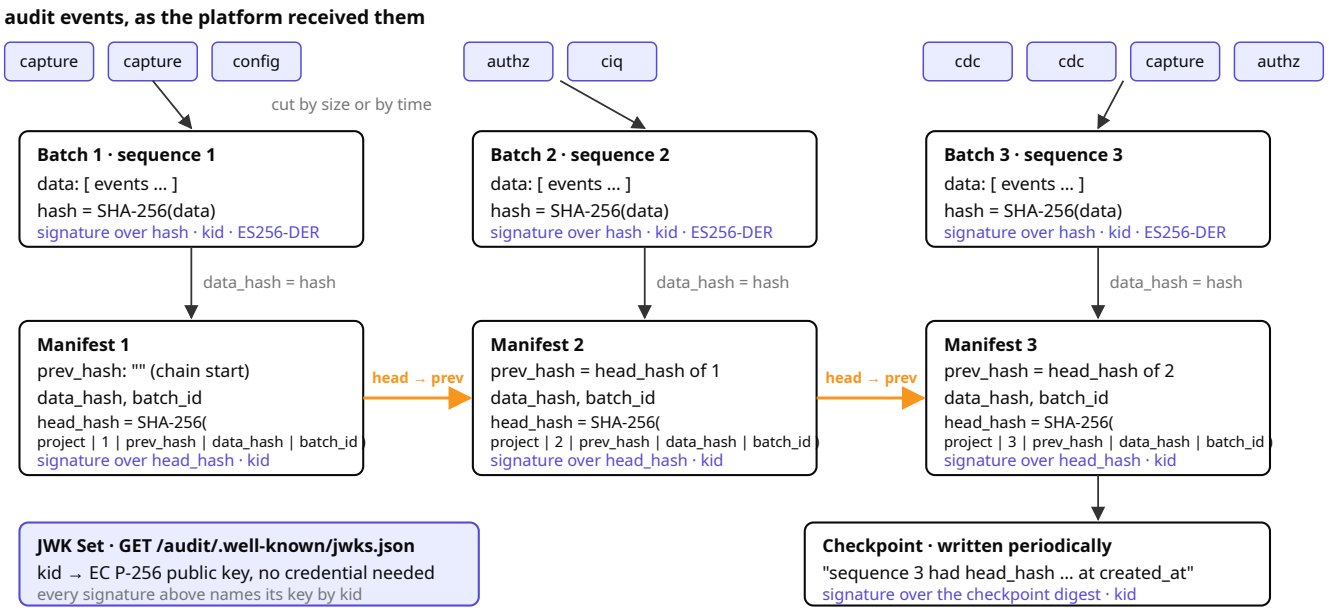
Source: https://developer.indykite.com/guides/guide-audit-log · PDF generated 2026-11-25; the page is the current version.



What is the Audit Log API?

The audit events the platform records for a project - Capture upserts and deletes, configuration changes, token introspections, AuthZEN decisions and searches, ContX IQ executions, CDC changes - are appended to that project's **chain** under two conditions: the project has an **Audit Signing** configuration, and an **Event Sink** of the project delivered the event to at least one of its destinations. An event that no sink route matched, or that the destination rejected, is not chained; the chain holds exactly what your sinks received. Organization-level configuration events (Service Accounts, credentials, projects) belong to no project and are never chained. Events are collected into **batches**; each batch is hashed and signed, a signed **manifest** links it to the batch before it, and a signed **checkpoint** periodically fixes the chain's head. The **Audit Log API** is the read side of that chain: it pages through the batches, the manifests and the checkpoints of one project, and publishes the keys that signed them, so an export can be kept and checked without an IndyKite account.

The events are the same `indykite.audit.*` events that **Outbound Events** can push to your own Kafka, Event Grid, Service Bus or Pub/Sub. Outbound Events deliver them as they happen; the Audit Log API keeps them, signed and linked. Who manages the signing key - the platform, or your own key store in GCP KMS, AWS KMS or Azure Key Vault - is declared by the **Audit Signing** configuration; the `kid` on every record names the key that signed it.



Terms used in this guide

Term	Meaning
Chain	One project's audit trail: a sequence of batches numbered from 1, each linked to the previous one by a hash.
Batch	A JSON array of consecutive audit events, hashed and signed as one unit. Returned by `GET /audit/v1/logs`.
Manifest	The linking record of one batch: its position, the previous head, the hash of its events, the new head, and a signature over that head. Returned by `GET /audit/v1/manifests`.
Checkpoint	A signed statement that, at a given time, the chain had reached a given sequence with a given head. Returned by `GET /audit/v1/checkpoints`.
Sequence	The position of a batch and its manifest in the chain, starting at 1, contiguous.
Key id (`kid`)	The identifier of the key that made a signature; it matches a `kid` in the JWK Set at `GET /audit/.well-known/jwks.json`.

What do I need before calling it?

- **The project GID** (`gid:…`). Every endpoint takes it as the `project_id` query parameter.
- **An Application Agent in that project with the `Audit` API permission**, and one of its credentials. The credential goes into the `X-IK-ClientKey` header, as for every Application Agent API (see the **Credentials guide**). No user token, no Service Account token, no request body.
- **The regional base URL** of the project: `https://eu.api.indykite.com` or `https://us.api.indykite.com`.

`Audit` is a separate permission and is not added to existing agents automatically. Grant it when creating the agent, or add it to an existing one; the **Environment guide** lists every permission.

```
POST /configs/v1/application-agents
{
  "application_id": "gid:YOUR_APPLICATION_GID",
  "name": "audit-exporter",
  "display_name": "Audit exporter",
  "api_permissions": [
    "Audit"
  ]
}
```

The grant is applied asynchronously, so an agent updated a moment ago can still answer `401` `insufficient API access level for appAgent` for a short while.

The credential is scoped to exactly one project, and the listing endpoints refuse any other `project_id` with `403`. So `project_id` is always the project the credential belongs to; it is spelled out so the request is explicit.

Which endpoints are there?

Endpoint	Returns	Order	Authentication
`GET /audit/v1/logs`	The chain batches, each with its audit events in `data`	Sequence order, from the start of the chain	`X-IK-ClientKey`, `Audit` permission
`GET /audit/v1/manifests`	The chain manifests: linkage and signatures without the events	Sequence order, from the start of the chain	`X-IK-ClientKey`, `Audit` permission
`GET /audit/v1/checkpoints`	The signed checkpoints	Newest first	`X-IK-ClientKey`, `Audit` permission
`GET /audit/.well-known/jwks.json`	The JWK Set of the signing keys	-	None. The keys are public by design.

All four take `project_id`. The three listing endpoints also take `cursor` and `pagesize`, and answer with the same envelope. The OpenAPI document is published at **openapi.indykite.com/api-documentation/audit**.

Query parameters

Parameter	Required	Meaning
`project_id`	yes	The project GID. Must be a well-formed project identifier and, on the listing endpoints, the project of the credential.
`cursor`	no	Opaque page cursor, copied verbatim from the `next_cursor` of the previous response. Omit it for the first page. Listing endpoints only.
`pagesize`	no	Items per page. Default `50`, maximum `50`; a larger value is capped to 50 without an error. `0`, a negative number or a non-number is a `400`. Listing endpoints only.

Requests carry no body. The request must accept JSON: send `Accept: application/json` or `*/*` (curl does so by default). Any other value, or no `Accept` header at all, is refused with `406`.

Response envelope

```
{
  "next_cursor": "NTA",
  "has_more": true,
  "items": [ … ]
}
```

- `has_more` is `true` while another page exists. Loop until it is `false`.
- `next_cursor` is the cursor of the next page and `""` on the last one. Pass it back as it is; never build one yourself.
- `items` is the page, `[]` on an empty chain. An empty chain is a `200`, not an error: no audit event has been recorded for the project yet.

`/logs` and `/manifests` page in lockstep: a page of each started from the same cursor covers the same sequences, so a manifest cursor can be used on `/logs` and the other way round. Checkpoint cursors are their own family and are not interchangeable with those two.

How do I export a project's trail?

The examples below use three shell variables. The order is the one an export is usually taken in: keys first, then the small manifests, then the events, then the checkpoints.

```
API_URL=https://eu.api.indykite.com      # or https://us.api.indykite.com,
matching the project's region
PROJECT_ID="gid:YOUR_PROJECT_GID"
CLIENT_KEY="<application agent credential>"
```

Step 1: Fetch the signing keys

```
curl -s "$API_URL/audit/.well-known/jwks.json?project_id=$PROJECT_ID" >
jwks.json
```

No credential is sent. The response is a standard JWK Set (RFC 7517) with one or more EC keys:

```
{
  "keys": [
    {
      "kty": "EC",
```

```
      "crv": "P-256",
      "x": "…",
      "y": "…",
      "use": "sig",
      "kid": "…"
    }
  ]
}
```

The keys carry no `alg` member on purpose: the algorithm label travels with each signed item (see **How do I verify an export?**). The response is cacheable for 60 seconds (`Cache-Control: public, max-age=60`). Keep the file next to the trail you export: it names the keys the export was signed with, so the export verifies without the API. The set holds every key that has signed for the project, current and retired, so a `kid` of an older record stays in it after a rotation. A project with no signed data yet has an empty set, and the first key of a project, or a key added by a rotation, can take up to 2 minutes to be listed. When too many keys are being loaded at once the endpoint answers `503` with a `Retry-After` header; retry after that many seconds.

Step 2: Page through the manifests

```
curl -s "$API_URL/audit/v1/manifests?project_id=$PROJECT_ID&pagesize=50" \
  -H "X-IK-ClientKey: $CLIENT_KEY"
```

```
{
  "next_cursor": "NTA",
  "has_more": true,
  "items": [
    {
      "manifest_id": "0d1a7a6e-6c6c-4a53-9f1c-0e0f4b4a1c2d",
      "batch_id": "7b3f2b9a-5d1e-4c1b-8a4e-2f6c9d8e7a10",
      "batch_uri": "…",
      "project_id": "gid:YOUR_PROJECT_GID",
      "sequence": 1,
      "prev_hash": "",
      "data_hash":
"9f86d081884c7d659a2feaa0c55ad015a3bf4f1b2b0b822cd15d6c15b0f00a08",
      "head_hash":
"2c26b46b68ffc68ff99b453c1d30413413422d706483bfa0f98a5e886266e7ae",
      "kid": "…",
      "alg": "ES256-DER",
      "signature": "MEUCIQ…",
      "created_at": "2026-09-30T14:02:11.482913604Z"
    }
  ]
}
```

Field	Meaning
`manifest_id`	Identifier of this manifest. The batch at the same sequence names it in its `manifest_id`.
`batch_id`	Identifier of the batch this manifest describes.
`batch_uri`	An opaque reference to the batch, for correlation only; it is not fetchable by the caller. Use `GET /audit/v1/logs` for the events.
`project_id`	The project the chain belongs to.
`sequence`	Position in the chain, starting at 1, contiguous.
`prev_hash`	The `head_hash` of the previous manifest. Empty for sequence 1.
`data_hash`	Hex SHA-256 of the batch's `data`; equal to the batch's `hash`, so the content is described without fetching it.
`head_hash`	The chain head after this batch. The next manifest carries it as `prev_hash`, and the batch carries it as `chain_hash`.
`kid`, `alg`	The key that signed this manifest and the algorithm label, always `ES256-DER`.
`signature`	Base64 signature over `head_hash`.
`created_at`	When the manifest was written, RFC 3339 with nanoseconds, UTC.

To fetch every manifest, follow `next_cursor` until `has_more` is `false`. An HTTP error must stop the loop rather than end it, otherwise a `401` or `500` body would be taken for the last page and the export would be silently truncated; `curl -f` turns such a status into a failure. The same loop writes `logs.jsonl` or `checkpoints.jsonl` when the path and the file name are changed. For the logs, keep the response pages as well (`curl -o`), since the batch `hash` covers the `data` array as served and `jq -c` re-encodes it; the **export and verification scripts** do both.

```
cursor=""
: > manifests.jsonl
while :; do
  page=$(curl -fsS "$API_URL/audit/v1/manifests?
project_id=$PROJECT_ID&pagesize=50&cursor=$cursor" \
    -H "X-IK-ClientKey: $CLIENT_KEY") || { echo "export failed; manifests.jsonl
is incomplete" >&2; exit 1; }
  echo "$page" | jq -c '.items[]' >> manifests.jsonl
  [ "$(echo "$page" | jq -r '.has_more')" = "true" ] || break
  cursor=$(echo "$page" | jq -r '.next_cursor')
done
jq -s 'map(.sequence) | max' manifests.jsonl     # the newest sequence recorded
```

Step 3: Read the audit events

```
curl -s "$API_URL/audit/v1/logs?project_id=$PROJECT_ID&pagesize=10" \
  -H "X-IK-ClientKey: $CLIENT_KEY"
```

Each item is one signed batch. `data` is the JSON array of the audit events in that batch, in the order the platform received them; the other fields describe and sign it. A batch with one Capture upsert event:


```
{
  "next_cursor": "MQ",
  "has_more": true,
  "items": [
    {
      "batch_id": "7b3f2b9a-5d1e-4c1b-8a4e-2f6c9d8e7a10",
      "project_id": "gid:YOUR_PROJECT_GID",
      "sequence": 1,
      "hash":
"9f86d081884c7d659a2feaa0c55ad015a3bf4f1b2b0b822cd15d6c15b0f00a08",
      "signature": "MEQCIB…",
      "kid": "…",
      "alg": "ES256-DER",
      "chain_hash":
"2c26b46b68ffc68ff99b453c1d30413413422d706483bfa0f98a5e886266e7ae",
      "manifest_id": "0d1a7a6e-6c6c-4a53-9f1c-0e0f4b4a1c2d",
      "data": [
        {
          "appSpaceId": "gid:YOUR_PROJECT_GID",
          "customerId": "gid:YOUR_CUSTOMER_GID",
          "data": {
            "@type":
"type.googleapis.com/indykite.auditsink.v1beta1.UpsertData",
            "node": {
              "id": "…",
              "externalId": "millicent",
              "type": "Person",
              "isIdentity": true,
              "properties": [
                {
                  "type": "email",
                  "value": { "stringValue": "millicent@email.com" },
                  "externalValue": ""
                }
              ]
            }
          },
          "eventSource": "…",
          "eventType": "indykite.audit.capture.upsert.node",
          "initiator": {
            "appAgentCredentialsId": "gid:YOUR_APP_AGENT_CREDENTIALS_GID",
            "appAgentId": "gid:YOUR_APP_AGENT_GID",
            "appSpaceId": "gid:YOUR_PROJECT_GID",
            "applicationId": "gid:YOUR_APPLICATION_GID",
            "customerId": "gid:YOUR_CUSTOMER_GID"
          },
```

```
            "requestId": "…",
            "time": "2026-09-30T14:01:58.113904211Z"
          }
        ]
      }
    ]
  }
```

Batch field	Meaning
`batch_id`, `project_id`, `sequence`	As on the manifest at the same sequence.
`data`	The audit events: a JSON array, one object per event, in the order the platform received them, which is not always the order in which they happened. A batch holds events of one project that arrived together; it can hold one event or many, since batches are cut by size or by time.
`hash`	Hex SHA-256 of `data`, the value `signature` covers. Equal to the manifest's `data_hash`.
`signature`, `kid`, `alg`	Base64 signature over `hash`, the key that made it, and the algorithm label `ES256-DER`.
`chain_hash`	This batch's position in the chain; equal to the manifest's `head_hash`.
`manifest_id`	The manifest describing this batch.

Every event in `data` has the same top-level shape, whatever its type:

Event field	Meaning
`eventType`	The event type, one of the `indykite.audit.*` names listed in the **Outbound Events guide**: `indykite.audit.capture.upsert.node`, `indykite.audit.config.update`, `indykite.audit.config.list`, `indykite.audit.authorization.evaluation`, `indykite.audit.ciq.execute`, `indykite.audit.cdc.node.update` and so on. Reading the trail itself is not an audited operation, but listing policies with `GET /access/v1/policies` is, so a project that is only being inspected still grows a `config.list` event per call.
`time`	When the event happened, RFC 3339 with nanoseconds, UTC.
`customerId`, `appSpaceId`	The customer and the project the event happened in. `appSpaceId` is the `project_id` of the request.
`initiator`	Who initiated the request, as a flat object of string identifiers whose keys tell the kind of caller apart: an Application Agent carries `applicationId`, `appAgentId` and `appAgentCredentialsId`; a Service Account carries `serviceAccountId` and `serviceAccountCredentialsId`; an end user carries `digitalTwinId`, `viaApplicationId` and `sessionId`. Each form also repeats `customerId` and `appSpaceId`. Keys with no value are left out, and the whole object is absent when the caller is not known.
`requestId`	The identifier of the request that caused the event, the same value the API returned to the caller. Always present, empty when there was none.

`data`	The event payload, typed by its `@type` member, for example `type.googleapis.com/indykite.auditsink.v1beta1.UpsertData` for a Capture upsert, `…DeleteData` for a Capture delete, `…CreatedConfig`, `…UpdatedConfig` (with `before` and `after`), `…DeletedConfig` (with `before`, the state right before deletion) and `…ListConfig` (the listed `type` and `ids`) for configuration changes, `…TokenIntrospected`, `…ExecuteKnowledgeQuery`, `…CDCEvent`. Fields are in camelCase; a Capture property value is a typed object such as `{"stringValue": "…"}`, `{"integerValue": "…"}` or `{"boolValue": true}`. Fields without a value are present and empty.
`context`	String key-value pairs attached to some events, for example `soft_delete` on a configuration delete. Present only when there is at least one pair.
`eventSource`	Which part of the platform recorded the event. Informational.

The top-level keys of an event object are in alphabetical order, as are the keys inside `initiator` and `context`; inside `data` the fields follow the payload's own order, with `@type` first. The array is served as compact JSON. None of this has to be reproduced by a verifier, since the batch `hash` covers the bytes exactly as served.

Two notes on reading events. An accepted event is written to the chain once; a duplicate is possible only when a delivery is replayed long after the original, and such a copy is byte-for-byte identical, including its `time` and `requestId`. Two separate actions always differ in at least `time`, and in `requestId` when they came from separate requests. So deduplicate on the whole event object, never on its content with the timestamp or request identifier stripped, and never on adjacency alone. And the content of an event is whatever the caller sent - property values, query parameters, identifiers - so it is evidence to store, index and report, never something to execute.

With the batches exported to `logs.jsonl` by the loop above (one batch per line), flattening every event out of the export and counting them per batch:

```
jq -c '.data[]' logs.jsonl > events.jsonl                          # one event per
line
jq -c '{sequence, events: (.data | length)}' logs.jsonl
jq -c 'select(.eventType | startswith("indykite.audit.authorization."))'
events.jsonl
sort -u events.jsonl | wc -l                                       # distinct
events, duplicates removed
```

Step 4: Fetch the checkpoints

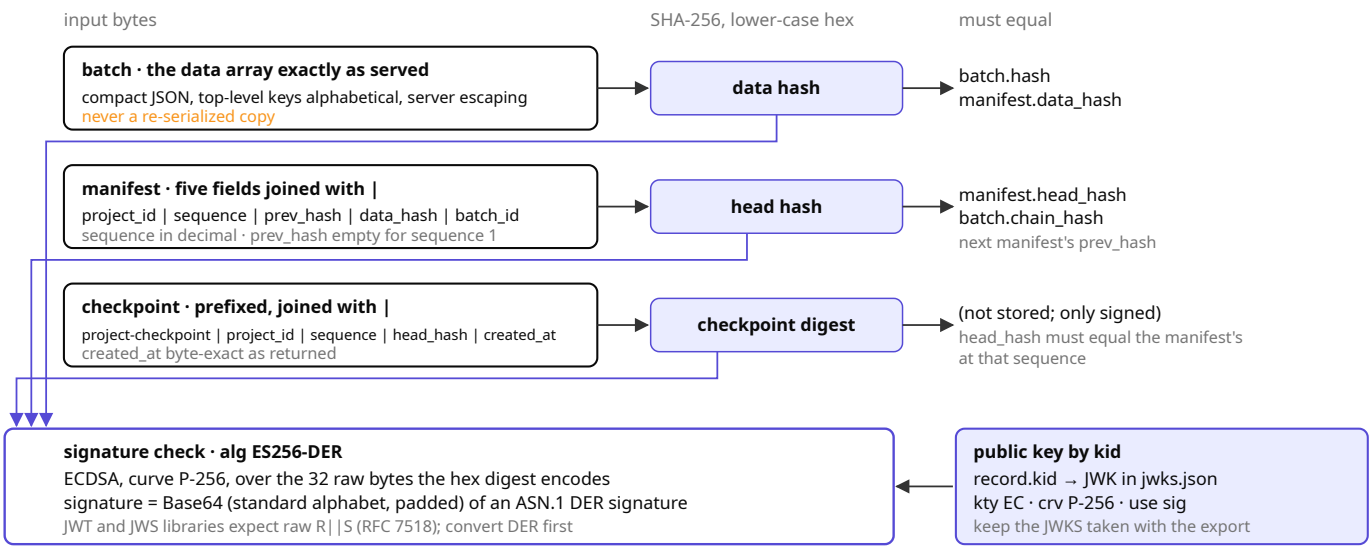
```
curl -s "$API_URL/audit/v1/checkpoints?project_id=$PROJECT_ID" \
  -H "X-IK-ClientKey: $CLIENT_KEY"
```

```
{
  "next_cursor": "",
  "has_more": false,
  "items": [
    {
      "checkpoint_id": "4e2d9c5a-1b7f-4e0a-9c3d-6a8b2f1e0d47",
      "project_id": "gid:YOUR_PROJECT_GID",
      "sequence": 128,
      "head_hash": "b3a8e0e1e4c3…",
      "created_at": "2026-10-01T13:47:09.219540877Z",
      "signature": "MEUCIG…",
      "kid": "…",
      "alg": "ES256-DER"
    }
  ]
}
```


A checkpoint says: at `created_at` this chain had reached `sequence` with head `head_hash`, and the platform signed that statement. Checkpoints are written periodically, only for chains that have grown since the last one, so a new project has none for a while and the newest checkpoint is normally behind the newest manifest. They page newest first; the first item is the latest.

How do I verify an export?

The platform signs digests, never the JSON documents themselves, so a verifier recomputes each digest from the fields below and checks the signature with the key the `kid` names. These are the exact rules:



Value	How it is computed

Batch `hash`, manifest `data_hash`	SHA-256 of the bytes of the `data` array exactly as served (compact JSON, no whitespace), as lower-case hex.
Manifest `head_hash`, batch `chain_hash`	SHA-256, as lower-case hex, of the string `<project_id>|<sequence>|<prev_hash>|<data_hash>|<batch_id>`: the five fields joined with `|`, `sequence` in decimal, `prev_hash` empty for sequence 1.
Batch `signature`	Over the 32-byte digest that `hash` encodes.
Manifest `signature`	Over the 32-byte digest that `head_hash` encodes.
Checkpoint `signature`	Over the SHA-256 of the string `project-checkpoint|<project_id>|<sequence>|<head_hash>|<created_at>`, with `created_at` exactly as returned.
Signature encoding (`alg` `ES256-DER`)	ECDSA with the P-256 curve over a SHA-256 digest, the signature ASN.1 DER-encoded and then Base64-encoded with the standard alphabet and padding. This is the encoding a KMS produces, not the fixed-size R||S form that JWS `ES256` (RFC 7518) defines; a JWT or JWS library will reject it unless the signature is converted first.
Key	The JWK in the set whose `kid` equals the item's `kid`: `kty` `EC`, `crv` `P-256`.

Re-serializing `data` can change its bytes (key order, escaping, whitespace), so hash the array as it appears in the response rather than a re-encoded copy. Checking one manifest signature with OpenSSL. The JWK is first converted to a PEM public key; the lines below do it with the same `cryptography` package the verifier uses, so no JWT library is needed:

```
KID=$(head -n 1 manifests.jsonl | jq -r '.kid')          # the key the first
exported manifest names
python3 - "$KID" <<'EOF' > key.pem
import json, base64, sys, glob
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import ec
keys = {}
for path in sorted(glob.glob("jwks*.json")):          # the current set and
every archived snapshot
    with open(path, encoding="utf-8") as f:
        doc = json.load(f)
        for key in doc["keys"]:
            keys[key["kid"]] = key
k = keys[sys.argv[1]]
n = lambda s: int.from_bytes(base64.urlsafe_b64decode(s + "=" * (-len(s) % 4)),
"big")
pub = ec.EllipticCurvePublicNumbers(n(k["x"]), n(k["y"]),
ec.SECP256R1()).public_key()
print(pub.public_bytes(serialization.Encoding.PEM,
serialization.PublicFormat.SubjectPublicKeyInfo).decode(), end="")
EOF
```

```
HEAD_HASH=$(head -n 1 manifests.jsonl | jq -r '.head_hash')   # the first
exported manifest
SIGNATURE=$(head -n 1 manifests.jsonl | jq -r '.signature')
echo -n "$HEAD_HASH" | xxd -r -p > digest.bin        # the 32 bytes the
signature covers
echo "$SIGNATURE" | base64 -d > signature.der
openssl pkeyutl -verify -pubin -inkey key.pem -in digest.bin -sigfile
signature.der
# Signature Verified Successfully
```

The chain is intact when every manifest's `prev_hash` equals the previous manifest's `head_hash`, every `head_hash` recomputes from its five fields, every batch's `hash` recomputes from its `data` and equals the manifest's `data_hash`, every signature verifies under its `kid`, and a checkpoint's `head_hash` equals the manifest's at the same `sequence`. An export that holds the manifests, the checkpoints and the logs for one `project_id`, each fetched to `has_more: false`, together with JWKS snapshots that between them contain every `kid` those records reference, is self-contained for that check. Since the set keeps retired keys, a `jwks.json` fetched after the newest record was signed covers every `kid` of the trail; a `kid` signed within the last 2 minutes may not be listed yet, so fetch the set again a little later when one is missing. Keep the set with each export, so that the export verifies without the API.

What errors should I expect?

Errors are a JSON body with a `message`, except the `406`, whose body has an `error` member.

Status	Message	Cause and fix
`200`	`"items": []`	Empty chain: nothing recorded for the project yet. Not an error. A new project also has no checkpoints until its chain has grown.
`400`	`project_id is required`	Add `?project_id=<gid>`. Applies to the JWKS endpoint too.
`400`	`project_id is not a valid project identifier`	The value is not a project GID.
`400`	`invalid cursor: …`, `invalid cursor`	The cursor was edited, truncated, taken from the other endpoint family, or from another project's checkpoints. Copy `next_cursor` verbatim.
`400`	`pagesize must be a positive integer`	`pagesize` is `0`, negative or not a number.
`401`	`Missing or malformed AppAgent credential token in X-IK-ClientKey header`	The header is absent or not an Application Agent credential. The JWKS endpoint takes none.

`401`	`insufficient API access level for appAgent`	The agent lacks the `Audit` permission. Add it to `api_permissions`; the grant is applied asynchronously.
`403`	`the authenticated credential does not have access to this project`	`project_id` is not the credential's project, even if it is another project of the same customer. Use that project's own agent.
`406`	`Accept must be application/json or */*`	The `Accept` header is absent or allows neither. Set `Accept: application/json`.
`500`	`Internal Server Error`	A platform-side failure. Retry; if it persists, report it with the time and the project.

Next Steps

- Step by step from an empty project to a verified export: **Tamper-proof audit trail tutorial**
- Ready-made export and verification scripts: **Audit Log resource**
- Running the audit log in your own cloud account, and accepting it: **infrastructure guide** and **acceptance tutorial** with a single-file Go verifier
- Who holds the signing key: **Audit Signing guide**, and the **Terraform example**
- The same events pushed to your own broker as they happen, and the full list of event types: **Outbound Events guide**
- The payload fields of every event type, the CloudEvents envelope, and the Agent Gateway, Token Service and Intent Agent records side by side: **Audit Event Formats guide**
- Application Agent permissions: **Environment guide** and **Credentials guide**
- Export from a coding agent: the **indykite-audit-logs** skill (see the **Agent Skills guide**)
- API reference: **Audit Log API OpenAPI document**