dstack: Events

(formerly known as Audit Logs)

Problem

Sometimes, when a run, instance, or any other resource changes its status, it may be very important to be able to see, without looking at server logs, at what time that event occurred, along with additional details, if any.

Examples

Major **The instance is unreachable (within X ms)** A run and idle/busy instance was terminated due to the instance becoming unreachable.	It's important to see when the corresponding instance becomes unreachable, when dstack terminates the run and instance, ideally along with important details (e.g., such as the provider's instance information). Note, it's also important that the run or instance status allows unambiguously interpreting the main reason (e.g. instead of "no capacity", say "terminated due to unreachable host").
Medium A run is retried (due to spot/other interruption, or no capacity)	It's important to see when the corresponding run changes the status due to the retry policy.
Medium A run generally changes status.	If a status changed, sometimes it may be important to know when exactly, e.g., to see when exactly the run failed or finished, etc. This can serve as a very convenient way to explore the history of job submissions.
Medium Charges to the balance	

And many others.

Overall, the following event categories are useful:

- Entity Create, Update, Delete operations performed by users (see approximate list in Appendix 1)
- Resource status changes
- Important entity property changes (instance health, probe status, deployment num)

- Important entity relationship changes (user added to project, job assigned to instance, volume attached to instance)
- Offer provisioning errors
- Backend errors in general (expired credentials, backend API down)
- Failed user actions, such as unauthorized access attempts

Solution

Main concepts

Events

Events are immutable persistent records about successful or failed actions performed by dstack users or dstack itself.

Actors

Each event is performed by an actor, which can be either a user or the system.

Targets

An event target is the entity directly affected by the event.

Each event has at least one target.
Example: User X registered.

Some events can have multiple targets of different types.
Example: User X joined Project Y.

Some events can have multiple targets of the same type.
Example (hypothetical): Project X set as parent for Project Y.

For a simpler UX, the following list of target types is suggested:

- Projects
- Users
- Backends
- Fleets
- Instances
- Runs
- Jobs
- Gateways
- Volumes
- Secrets

- Repos (?)
- File archives (?)

While dstack has more entities internally, actions on those entities can be represented as events targeting the entities from the list above. For example, action on placement groups can be represented as events targeting fleets, and actions on volume attachments can be represented as events targeting volumes and instances.

Access control

Global admins can view all events.

Other users can view:

- events targeting entities in the projects the user is a member of
- events targeting the user itself

API

Listing events

Route

`POST /api/events/list`

Response

The response includes the list of events with their details and targets. Example:

```
[
    {
        "id": "some-event-uuid",
        "recorded_at": "1970-01-01T00:00:01",
        "message": "User deleted",
        "actor_user_id": "actor-user-uuid",
        "actor_user_name": "actor-user",
        "targets": [
            {
                "type": "user",
                "project_id": null,  // users are not bound to a project
                "project_name": null,
                "id": "deleted-user-uuid",
                "name": "deleted-user"
            }
        ]
    },
    {
```

```
        "id": "other-event-uuid",
        "recorded_at": "1970-01-01T00:00:02",
        "message": "Volume attached to instance",
        "actor_user_id": null,  // system event
        "actor_user_name": null,
        "targets": [
            {
                "type": "volume",
                "project_id": "some-project-uuid",
                "project_name": "some-project",
                "id": "some-volume-uuid",
                "name": "some-volume"
            },
            {
                "type": "instance",
                "project_id": "some-project-uuid",
                "project_name": "some-project",
                "id": "some-instance-uuid",
                "name": "some-instance"
            },
            {
                "type": "job",
                "project_id": "some-project-uuid",
                "project_name": "some-project",
                "volume_id": "some-job-uuid",
                "volume_name": "some-job"
            }
        ]
    }
]
```

All ID fields are followed by name fields for user-friendly display in the UI and CLI.

Request filters

Users will need to filter events based on various properties.

Examples of useful queries:

- All events visible to the current user
- All events within project X
- All events about actions on project X (project creation, deletion, membership updates, etc)
- All events about actions on secrets (secret creation, update, deletion) within project X
- All events about actions performed by user X
- All events about actions performed by user X within project Y
- All events about actions on volumes (volume creation, deletion) performed by user X within project Y

Below are the suggested filters to support these and other queries.

Target container filters

These filters allow to include only those events that have a target in the specified container. For example, `within_fleets: [my-fleet]` will include events about the fleet `my-fleet` and all instances in that fleet.

- `within_projects: list[UUID]`
- `within_fleets: list[UUID]`
- `within_runs: list[UUID]`

Target filters

These filters allow to include only those events that **directly** target the specified entity. For example, `target_fleets: [my-fleet]` will include events about the fleet `my-fleet`, but **not** about instances in that fleet.

- `target_projects: list[UUID]`
- `target_users: list[UUID]`
- `target_backends: list[UUID]`
- `target_fleets: list[UUID]`
- `target_instances: list[UUID]`
- `target_runs: list[UUID]`
- `target_jobs: list[UUID]`
- `target_gateways: list[UUID]`
- `target_volumes: list[UUID]`
- `target_repos: list[UUID]`
- `target_secrets: list[UUID]`

Target type filter

This filter allows to include only those events that target at least one of the specified entity types. For example, `include_target_types: [secret]` will include only events targeting secrets.

If an event targets several entity types, it is included in the response if at least one of its targets matches `include_target_types`.

- `include_target_types: list['project' | 'user' | 'backend' | 'fleet' | 'instance' | 'run' | 'job' | 'gateway' | 'volume' | 'repo']`

Note on naming: the filter name has the `include_` prefix to highlight that its semantics are different from `target_projects`, `target_fleets`, etc.

Actor filter

This filter allows to include only those events that represent actions performed by specific actors (users or the system).

- `actors: list[UUID | null]`

`null` represents the system actor.

Pagination filters

Same as in other API routes.

- `prev_id: UUID`
- `prev_recorded_at: datetime`
- `limit: int`
- `ascending: bool`

UI

Events page

The new `Events` page will be added to display the list of events.

The events table has these columns:

- **Recorded At** — date and time with second-level precision.
- **Actor** — "-" for system events, username with hyperlink to user settings for events with a user actor.
- **Targets** — newline-separated list of event targets. Each target is formatted as: `<target-type> <project-name>/<target-name>`,
  where:
  - `<project-name>/` is only included if it is not null and the target type is not "project";
  - `<project-name>` is a hyperlink to project settings;
  - `<target-name>` is a hyperlink to target details if target type is one of: project, user, fleet, run.
- **Message**.

Ordered by Recorded At descending.

Example:

Recorded At	Actor	Targets	Message
01/01/1970 00:00:05	actor-user	fleet new-project/new-fleet	Fleet created
01/01/1970 00:00:04	actor-user	gateway new-project/new-gateway	Gateway created
01/01/1970 00:00:03	actor-user	project new-project	Project created
01/01/1970 00:00:02	-	volume some-project/some-volume instance some-project/some-instance job some-project/some-job	Volume attached to instance
01/01/1970 00:00:01	actor-user	user deleted-user	User deleted

We'll be able to provide hyperlinks for more target types (volumes, instances, jobs, etc) later, once we add the relevant APIs and pages.

Controls and behaviors

- Infinite scrolling to view older events.
- A refresh button to load newer events.
- A filter field that covers the API filters.

Resource detail pages

Resource detail pages will be updated to include references (buttons or hyperlinks) that allow opening the `Events` page with filters pre-set to view the events related to that resource.

The following references can be added on the following pages:

- Run details page:
  - View run events (the `target_runs` filter)
  - View events within the run (the `within_runs` filter)
- Job details page:
  - View job events (the `target_jobs` filter)
- Fleet details page:
  - View fleet events (the `target_fleets` filter)
  - View events within the fleet (the `within_fleets` filter)
- Project settings page:
  - View project events (the `target_projects` filter)
  - View events within the project (the `within_projects` filter)
- User settings page:
  - View user events (the `target_users` filter)
  - View user activity (the `actors` filter)

CLI

The CLI will get the `dstack event` command for listing events. It will accept the same filters as the API does, except it can accept resource names and resolve them to IDs on behalf of the user.

```
$ dstack event --within-fleet my-fleet

RECORDED AT          TARGET                MESSAGE
2025-11-07 00:00:00  instance my-fleet-0   Instance unreachable
2025-11-07 00:00:01  instance my-fleet-1   Volume attached to instance
                     volume my-volume
                     job my-job
2025-11-07 00:00:02  instance my-fleet-2   Instance terminated due to idle duration
```

`dstack event` can have a `--watch` switch to track events interactively. It will be implemented by requesting the new event batch once every N seconds.

`dstack apply` can track and display events related to the resource currently being provisioned.

Data model

The `events` table entries hold details about an event and its actor.

```
                          Table "events"
   Column    |            Type             | Collation | Nullable | Default
-------------+-----------------------------+-----------+----------+---------
 id          | uuid                        |           | not null |
 message     | text                        |           | not null |
 recorded_at | timestamp without time zone |           | not null |
 user_id     | uuid                        |           |          |
Indexes:
    "pk_events" PRIMARY KEY, btree (id)
    "ix_events_recorded_at" btree (recorded_at)
    "ix_events_user_id" btree (user_id)
```

`The event_targets` table entries contain generic details about an event target — its type (`run`, `job`, `instance`, etc), ID, name (denormalized for performance reasons), parent project ID (except when `type = 'user'`).

```
                        Table "event_targets"
      Column       |       Type        | Collation | Nullable | Default
-------------------+-------------------+-----------+----------+---------
 id                | uuid              |           | not null |
 event_id          | uuid              |           | not null |
 entity_project_id | uuid              |           |          |
 entity_type       | character varying |           | not null |
 entity_name       | character varying |           | not null |
 entity_id         | uuid              |           | not null |
Indexes:
    "event_targets_pkey" PRIMARY KEY, btree (id)
    "ix_event_targets_entity_id" btree (entity_id)
    "ix_event_targets_entity_project_id" btree (entity_project_id)
    "ix_event_targets_entity_type" btree (entity_type)
    "ix_event_targets_event_id" btree (event_id)
```

An alternative normalized data schema is described in Appendix 3, but not suggested due to worse performance and maintainability.

Performance testing

PostgreSQL setup:

```
# SELECT COUNT(*) FROM events;
1830252
# SELECT COUNT(*) FROM event_targets;
1830312
# SELECT COUNT(*) FROM projects;
8
# SELECT COUNT(*) FROM users;
61
# SELECT COUNT(*) FROM backends;
65
# SELECT COUNT(*) FROM fleets;
9000
# SELECT COUNT(*) FROM instances;
630000
# SELECT COUNT(*) FROM runs;
600000
# SELECT COUNT(*) FROM jobs;
1200000
# SELECT COUNT(*) FROM gateways;
60
# SELECT COUNT(*) FROM volumes;
30000
# SELECT COUNT(*) FROM repos;
```

```
6
```

Selecting **all** events (**not realistic**; pagination will be enforced in real conditions) and their targets.

```
SELECT
    events.id,
    events.message,
    event_targets.entity_type,
    event_targets.entity_name,
    event_targets.entity_id,
    projects.name,
    users.name
FROM events
JOIN event_targets ON events.id = event_targets.event_id
LEFT OUTER JOIN projects ON event_targets.entity_project_id = projects.id
JOIN users ON events.user_id = users.id;

<...>

(1830192 rows)

Time: 7743.988 ms (00:07.744)
```

^Note: if necessary, the schema can be further denormalized to avoid joining with projects and users.

Selecting **1000** most recent events and their targets.

```
SELECT
    events.id,
    events.message,
    event_targets.entity_type,
    event_targets.entity_name,
    event_targets.entity_id,
    projects.name,
    users.name
FROM events
JOIN event_targets ON events.id = event_targets.event_id
LEFT OUTER JOIN projects ON event_targets.entity_project_id = projects.id
JOIN users ON events.user_id = users.id
WHERE events.id IN (
    SELECT id FROM (
        SELECT DISTINCT events.id AS id, events.happened_at
        FROM events
        JOIN event_targets ON events.id = event_targets.event_id
        ORDER BY events.happened_at DESC
```

```
        LIMIT 1000
    )
);

<...>

(1000 rows)

Time: 1027.013 ms (00:01.027)  <-- Cold cache
Time: 27.244 ms                <-- Hot cache
```

Selecting **100** most recent events and their targets.

```
SELECT
    events.id,
    events.message,
    event_targets.entity_type,
    event_targets.entity_name,
    event_targets.entity_id,
    projects.name,
    users.name
FROM events
JOIN event_targets ON events.id = event_targets.event_id
LEFT OUTER JOIN projects ON event_targets.entity_project_id = projects.id
JOIN users ON events.user_id = users.id
WHERE events.id IN (
    SELECT id FROM (
        SELECT DISTINCT events.id AS id, events.happened_at
        FROM events
        JOIN event_targets ON events.id = event_targets.event_id
        ORDER BY events.happened_at DESC
        LIMIT 100
    )
);

<...>

(100 rows)

Time: 233.296 ms  <-- Cold cache
Time: 5.131 ms    <-- Hot cache
```

Selecting **100** most recent events **filtered by target type**.

```
SELECT
    events.id,
    events.message,
    event_targets.entity_type,
```

```
    event_targets.entity_name,
    event_targets.entity_id,
    projects.name,
    users.name
FROM events
JOIN event_targets ON events.id = event_targets.event_id
LEFT OUTER JOIN projects ON event_targets.entity_project_id = projects.id
JOIN users ON events.user_id = users.id
WHERE events.id IN (
    SELECT id FROM (
        SELECT DISTINCT events.id AS id, events.happened_at
        FROM events
        JOIN event_targets ON events.id = event_targets.event_id
        WHERE event_targets.entity_type = 'job'
        ORDER BY events.happened_at DESC
        LIMIT 100
    )
);

<...>

(100 rows)

Time: 336.186 ms  <-- Cold cache
Time: 4.823 ms    <-- Hot cache
```

Selecting 100 most recent events filtered by **another target type**.

```
SELECT
    events.id,
    events.message,
    event_targets.entity_type,
    event_targets.entity_name,
    event_targets.entity_id,
    projects.name,
    users.name
FROM events
JOIN event_targets ON events.id = event_targets.event_id
LEFT OUTER JOIN projects ON event_targets.entity_project_id = projects.id
JOIN users ON events.user_id = users.id
WHERE events.id IN (
    SELECT id FROM (
        SELECT DISTINCT events.id AS id, events.happened_at
        FROM events
        JOIN event_targets ON events.id = event_targets.event_id
        WHERE event_targets.entity_type = 'project'
        ORDER BY events.happened_at DESC
        LIMIT 100
    )
```

```
);

(126 rows)

Time: 133.915 ms  <-- Cold cache
Time: 4.107 ms    <-- Hot cache
```

Ingestion rate approximation

We can approximate the size of a single event and a single event target based on the relation sizes with test data from the previous section.

The event size depends on the message size, which varies with each event. Since the test data mostly consisted of short messages, we'll add an additional 40 bytes to account for longer, real-world messages.

```
SELECT pg_total_relation_size('events') / (SELECT COUNT(*) FROM events) + 40
AS approx_event_size;
 approx_event_size
-------------------
               370
(1 row)

SELECT pg_total_relation_size('event_targets') / (SELECT COUNT(*) FROM
event_targets) AS approx_event_target_size;
 approx_event_target_size
--------------------------
                      248
(1 row)
```

Estimating the number of events is more challenging, as it depends on the specific dstack setup and usage pattern. However, we can assume that most events will relate to runs, jobs, and instances.

A single-job run with a newly provisioned instance and a volume might result in the following events:

1. Run submitted
2. Run provisioning
3. Run running
4. Run terminating
5. Run terminated
6. Job submitted
7. Job provisioning
8. Job pulling
9. Job running
10. Job terminating

11. Job terminated
12. Instance created (2 targets)
13. Instance busy
14. Instance idle
15. Instance terminating
16. Instance terminated
17. Volume attached (3 targets)
18. Volume detached (3 targets)
19. File archive uploaded

19 events * 370 bytes + 24 targets * 248 bytes = 12982 bytes **(~12.7 KiB) per run**

Note: This is a very rough approximation, as some real-world runs could involve orders of magnitude more events, particularly when they include multiple jobs, autoscaling, etc. Exceptional events, such as offer provisioning errors or backend errors, are also likely to require significantly more space due to long error descriptions.

Miscellaneous considerations

- Events have a TTL configurable on the server level. The default is to be determined later, but can likely be several weeks or months.
- Event records are unaffected by entity deletions. It can be useful to see events targeting a deleted entity.
- Several API routes may need to be updated or added to be able to return resource IDs and find resources by IDs. This will allow linking event targets to resource details pages in the UI.
- Events are written in the same database transaction as the rest of the changes. If the transaction fails and the resource is not updated, the event is not written to the database.
- `events.recorded_at` is set on the database side (using `server_default` or `func.now()`) rather than the dstack server side. That way, event write order will (hopefully) match the `recorded_at` order, so events won't be lost during pagination when tracking with `dstack event --watch`..

Implementation plan

1. Implement the event storage and emission framework.
2. Add events for some target types (possibly fleets and instances, or runs and jobs).
3. Implement the event listing API.
4. Implement the CLI.
5. Add events for more target types.
6. Add/update APIs required for linking events to resource pages in the UI.
7. Implement the UI.

Appendix 1. Useful Create-Update-Delete events

- User created
- User updated
- User SSH key refreshed
- User token refreshed
- User deleted
- Project created
- Project updated
- Project deleted
- Project member added
- Project member removed
- Backend created
- Backend updated
- Backend deleted
- Fleet created
- Fleet updated
- Fleet termination started
- Instance termination started
- Instance deleted
- Repo created?
- Repo updated?
- Repo deleted?
- Repo creds created?
- Repo creds updated?
- Repo creds deleted?
- Repo code uploaded?
- Run created
- Run updated
- Run stopped
- Run deleted
- Secret created
- Secret updated
- Secret deleted
- Gateway created
- Gateway updated
- Gateway deleted
- Volume created
- Volume deleted
- File archive uploaded?

Appendix 2. Events currently logged with logger.info

Module	Message
Backends	Skipping EBS volume %s detach since it's already detached
	Requesting %s %s instance in %s...
	Request succeeded
	Launching %s gateway instance in %s...
	Request succeeded
	Skipping unknown TPU: %s
	Skipping TPU due to invalid number of tensor cores: %s
	Skipping unknown TPU: %s
Server lifespan	Initializing the default configuration...
	Initialized the default configuration at [link=file://{SERVER_CONFIG_FILE_PATH}]{server_config_dir}[/link]
	Applying [link=file://{SERVER_CONFIG_FILE_PATH}]{server_config_dir}[/link]...
	Background processing is disabled
	Job network mode: %s (%d)
	The admin token is {admin.token.get_plaintext_or_error()}
	The dstack server {dstack_version} is running at {SERVER_URL}
	No server/config.yml. Skipping encryption configuration.
	Found not enabled plugin %s. Plugin will not be loaded.
	Found not enabled builtin plugin %s. Plugin will not be loaded.
	Loaded plugin %s
	Connecting to {len(gateway_computes)} gateways...
Lifespan + background	Gateway %s configured

Background processing	Connection to gateway %s is down, restarting
	Terminated compute group %s
	Terminating instance %s: %s
	Added %s instances to fleet %s
	Fleet %s deleted due to deleted project
	Automatic cleanup of an empty fleet %s
	Fleet %s deleted
	%s: gateway status has changed %s -> %s%s
	%s: started gateway provisioning
	%s: failed to create gateway compute: %r
	Deleting idle volume %s
	Deleted idle volume %s
	Instance %s idle duration expired: idle time %ss. Terminating
	Adding ssh instance %s...
	The instance %s (%s) was successfully added
	Connected to {remote_details.ssh_user} {remote_details.host}
	%s: CPU arch is %s
	Created instance %s
	Terminated instance %s: %s
	Detected busy instance %s with finished job. Marked as TERMINATING
	Instance %s has switched to %s status
	Instance %s terminated
	Created placement group %s in %s/%s
	Placement group %s is still in use. Skipping deletion for now.
	Deleted placement group %s

	%s: terminating due to secrets interpolation error
	"%s: now is %s", fmt(job_model), job_model.status.name
	"%s: non-zero exit status %s", fmt(job_model), exit_status
	"%s: GPU utilization check: terminating", fmt(job_model)
	%s: run status has changed PENDING -> SUBMITTED
	%s: run status has changed %s -> %s
	%s: run took %.2f seconds from submission to provisioning.
	"%s: provisioned %s new instance(s)", fmt(job_model), len(provisioned_jobs
	Created a new instance %s for job %s
	The job %s switched instance %s status to BUSY
	"%s: now is provisioning on '%s'", fmt(job_model), instance.name
	Attaching volumes: %s
	Started submitted volume %s processing
	Registering external volume %s
	Provisioning new volume %s
	Failed to create volume %s: %s
	Added new volume %s
	%s: run status has changed TERMINATING -> %s, reason: %s
	%s: scaling %s %s replica(s)
	Detaching volumes: %s
	%s: instance '%s' has been released, new status is %s
	%s: job status is %s, reason: %s
	Force detaching volume %s from %s
	failed to collect metrics for task %s: %s
	%s: service is registered as %s
	%s: service replica registered to receive requests, gateway=%s
	%s: service replica unregistered from receiving requests, gateway=%s

Services	Deleting fleets: %s
	Deleted repos %s in project %s
	Deleted repo creds for repo %s user %s
	Deleted secrets %s in project %s
	Deleted users %s by user %s
	Deleting volumes: %s
Services	Deleting gateways: %s
	Deleting gateway compute for %s...
	Deleted gateway compute for %s
	Gateway %s updated
Services + Lifespan	Deleted backends %s in project %s
Services + Background	Requesting instance offers from backends: %s

Appendix 3. Alternative data model

This alternative data model includes a normalized sparse non-generic targets table, where each target type has its own ID column, and only one `target_*_id` column is expected to be set for each entry. This approach allows joining the targets table with all entity tables at once to fetch live data such as entity names.

```
                          Table "events"
   Column    |            Type             | Collation | Nullable | Default
-------------+-----------------------------+-----------+----------+---------
 id          | uuid                        |           | not null |
 message     | text                        |           | not null |
 recorded_at | timestamp without time zone |           | not null |
 user_id     | uuid                        |           |          |
Indexes:
    "pk_events" PRIMARY KEY, btree (id)
    "ix_events_recorded_at" btree (recorded_at)
```

```
    "ix_events_user_id" btree (user_id)
```

```
                Table "event_targets"
       Column        | Type | Collation | Nullable | Default
---------------------+------+-----------+----------+---------
 id                  | uuid |           | not null |
 event_id            | uuid |           | not null |
 within_project_id   | uuid |           |          |
 target_project_id   | uuid |           |          |
 target_user_id      | uuid |           |          |
 target_backend_id   | uuid |           |          |
 target_fleet_id     | uuid |           |          |
 target_instance_id  | uuid |           |          |
 target_run_id       | uuid |           |          |
 target_job_id       | uuid |           |          |
 target_gateway_id   | uuid |           |          |
 target_volume_id    | uuid |           |          |
 target_repo_id      | uuid |           |          |
Indexes:
    "pk_event_targets" PRIMARY KEY, btree (id)
    "ix_event_targets_event_id" btree (event_id)
    "ix_event_targets_target_backend_id" btree (target_backend_id)
    "ix_event_targets_target_fleet_id" btree (target_fleet_id)
    "ix_event_targets_target_gateway_id" btree (target_gateway_id)
    "ix_event_targets_target_instance_id" btree (target_instance_id)
    "ix_event_targets_target_job_id" btree (target_job_id)
    "ix_event_targets_target_project_id" btree (target_project_id)
    "ix_event_targets_target_repo_id" btree (target_repo_id)
    "ix_event_targets_target_run_id" btree (target_run_id)
    "ix_event_targets_target_user_id" btree (target_user_id)
    "ix_event_targets_target_volume_id" btree (target_volume_id)
    "ix_event_targets_within_project_id" btree (within_project_id)
```

Performance testing

PostgreSQL setup:

```
# SELECT COUNT(*) FROM events;
1830252
# SELECT COUNT(*) FROM event_targets;
1830312
# SELECT COUNT(*) FROM projects;
8
# SELECT COUNT(*) FROM users;
```

```
61
# SELECT COUNT(*) FROM backends;
65
# SELECT COUNT(*) FROM fleets;
9000
# SELECT COUNT(*) FROM instances;
630000
# SELECT COUNT(*) FROM runs;
600000
# SELECT COUNT(*) FROM jobs;
1200000
# SELECT COUNT(*) FROM gateways;
60
# SELECT COUNT(*) FROM volumes;
30000
# SELECT COUNT(*) FROM repos;
6
```

Joining **all** events (significantly more than in real conditions, where pagination will be enforced), event targets, and targeted entities:

```
SELECT
    events.id,
    events.message,
    projects.name,
    users.name,
    backends.type,
    fleets.name,
    instances.name,
    runs.run_name,
    jobs.job_name,
    gateways.name,
    volumes.name,
    repos.name
FROM events
JOIN event_targets ON events.id = event_targets.event_id
LEFT OUTER JOIN projects ON event_targets.target_project_id = projects.id
LEFT OUTER JOIN users ON event_targets.target_user_id = users.id
LEFT OUTER JOIN backends ON event_targets.target_backend_id = backends.id
LEFT OUTER JOIN fleets ON event_targets.target_fleet_id = fleets.id
LEFT OUTER JOIN instances ON event_targets.target_instance_id = instances.id
LEFT OUTER JOIN runs ON event_targets.target_run_id = runs.id
LEFT OUTER JOIN jobs ON event_targets.target_job_id = jobs.id
LEFT OUTER JOIN gateways ON event_targets.target_gateway_id = gateways.id
LEFT OUTER JOIN volumes ON event_targets.target_volume_id = volumes.id
LEFT OUTER JOIN repos ON event_targets.target_repo_id = repos.id;

<...>
```

```
(1830312 rows)

Time: 7277.107 ms (00:07.277)
```

Joining a **~1000 entry subset** of events, all event targets, and targeted entities.

```
SELECT
    events.id,
    events.message,
    projects.name,
    users.name,
    backends.type,
    fleets.name,
    instances.name,
    runs.run_name,
    jobs.job_name,
    gateways.name,
    volumes.name,
    repos.name
FROM events
JOIN event_targets ON events.id = event_targets.event_id
LEFT OUTER JOIN projects ON event_targets.target_project_id = projects.id
LEFT OUTER JOIN users ON event_targets.target_user_id = users.id
LEFT OUTER JOIN backends ON event_targets.target_backend_id = backends.id
LEFT OUTER JOIN fleets ON event_targets.target_fleet_id = fleets.id
LEFT OUTER JOIN instances ON event_targets.target_instance_id = instances.id
LEFT OUTER JOIN runs ON event_targets.target_run_id = runs.id
LEFT OUTER JOIN jobs ON event_targets.target_job_id = jobs.id
LEFT OUTER JOIN gateways ON event_targets.target_gateway_id = gateways.id
LEFT OUTER JOIN volumes ON event_targets.target_volume_id = volumes.id
LEFT OUTER JOIN repos ON event_targets.target_repo_id = repos.id
WHERE recorded_at > TIMESTAMP '2025-11-04 23:59:13.000000+00';

<...>

(1018 rows)

Time: 1215.187 ms (00:01.215)
```

Joining a **~100 entry subset** of events, all event targets, and targeted entities.

```
SELECT
    events.id,
    events.message,
    projects.name,
    users.name,
    backends.type,
    fleets.name,
```

```
    instances.name,
    runs.run_name,
    jobs.job_name,
    gateways.name,
    volumes.name,
    repos.name
FROM events
JOIN event_targets ON events.id = event_targets.event_id
LEFT OUTER JOIN projects ON event_targets.target_project_id = projects.id
LEFT OUTER JOIN users ON event_targets.target_user_id = users.id
LEFT OUTER JOIN backends ON event_targets.target_backend_id = backends.id
LEFT OUTER JOIN fleets ON event_targets.target_fleet_id = fleets.id
LEFT OUTER JOIN instances ON event_targets.target_instance_id = instances.id
LEFT OUTER JOIN runs ON event_targets.target_run_id = runs.id
LEFT OUTER JOIN jobs ON event_targets.target_job_id = jobs.id
LEFT OUTER JOIN gateways ON event_targets.target_gateway_id = gateways.id
LEFT OUTER JOIN volumes ON event_targets.target_volume_id = volumes.id
LEFT OUTER JOIN repos ON event_targets.target_repo_id = repos.id
WHERE recorded_at > TIMESTAMP '2025-11-04 23:59:54.000000+00';

<...>

(118 rows)

Time: 37.906 ms
```

Selecting 100 most recent events **filtered by target type**.

```
SELECT
    events.id,
    events.message,
    projects.name,
    users.name,
    backends.type,
    fleets.name,
    instances.name,
    runs.run_name,
    jobs.job_name,
    gateways.name,
    volumes.name,
    repos.name
FROM events
JOIN event_targets ON events.id = event_targets.event_id
LEFT OUTER JOIN projects ON event_targets.target_project_id = projects.id
LEFT OUTER JOIN users ON event_targets.target_user_id = users.id
LEFT OUTER JOIN backends ON event_targets.target_backend_id = backends.id
LEFT OUTER JOIN fleets ON event_targets.target_fleet_id = fleets.id
LEFT OUTER JOIN instances ON event_targets.target_instance_id = instances.id
LEFT OUTER JOIN runs ON event_targets.target_run_id = runs.id
```

```
LEFT OUTER JOIN jobs ON event_targets.target_job_id = jobs.id
LEFT OUTER JOIN gateways ON event_targets.target_gateway_id = gateways.id
LEFT OUTER JOIN volumes ON event_targets.target_volume_id = volumes.id
LEFT OUTER JOIN repos ON event_targets.target_repo_id = repos.id
WHERE events.id IN (
    SELECT id FROM (
        SELECT DISTINCT events.id AS id, events.recorded_at
        FROM events
        JOIN event_targets ON events.id = event_targets.event_id
        WHERE event_targets.target_project_id IS NOT NULL
        ORDER BY events.recorded_at DESC
        LIMIT 100
    )
);

<...>

(126 rows)

Time: 3623.941 ms (00:03.624)
```