Simple Sync

Overview

Events

Simple Sync is a data storage system which uses the Event Sourcing pattern. It is meant to be optimized for usage as a backend for local-first apps.

Data is represented as a sequence of events.

Each event has the following schema of 6 fields, represented as a JSON object.

```
{
  "uuid": "string",
  "timestamp": "uint64",
  "user": "string",
  "item": "string",
  "action": "string",
  "payload": "string"
}
```

Validation

1. All events are evaluated against the ACL.
2. The UUID must be a valid v7 UUID.
3. The timestamp must be a valid 64 bit unsigned integer representing the number of milliseconds since the epoch.
4. The timestamp must match the timestamp value encoded in the UUID.
5. The user, item, and action may contain any of the following characters:
    - Lowercase letters
    - Uppercase letters
    - Numbers
    - The following punctuation: `.`, `/`, `:`, `-`, and `_`
        - `.` is preferred as a separator between segments, for example: `admin.123` rather than `admin-123`.
6. Users, items, and actions that begin with `.` are **reserved** for internal usage and subject to additional validation.
7. The payload must be a valid JSON object encoded as a string.

Any event that fails validation is **not** added to the authoritative history.

Data Querying

All data querying is handled locally. This means that Simple Sync is inappropriate for situations that require millions of items. It is much better suited for systems that need to store a small amount of items that don't change too frequently.

Syncing Process

The syncing process ensures that all clients have the latest version of the event history. Here's a step-by-step breakdown:

1. **Authoritative History:** The server maintains the authoritative history of all events. This is the single source of truth for the data.
2. **Initial Sync:** When a new client comes online, it performs an initial sync by retrieving the entire authoritative history from the server via the API.
3. **Local Events:** After the initial sync, the client keeps a local history of any events generated by the user on that client. These events are stored locally and represent changes that have not yet been synchronized with the server.
4. **Periodic Push:** The client periodically pushes its local events to the server via the API. This is typically triggered by the user (e.g., with a "sync" button) to avoid conflicts during active use.
5. **Server-Side Merge:** The server receives the incoming events from the client and merges them into the authoritative history, applying any necessary conflict resolution or validation logic. The server will also ensure that all events follow the rules defined in the ACL Specification.
6. **New Authoritative History:** The server responds to the client's push request with the updated authoritative history.
7. **Client Update:** The client replaces its local copy of the authoritative history with the new version received from the server.

This process ensures that all clients eventually converge on the same state, while also allowing for offline work and local data access.

v1 API

Authentication

All endpoints (except `/api/v1/setup/exchangeToken` and `/api/v1/health`) require authentication via API key passed in the `Authorization` header as `Bearer <API_KEY>`.

API keys are obtained through the setup token exchange process:

1. An admin generates a setup token for a user via `POST /api/v1/user/generateToken`
2. The user exchanges the setup token for an API key via `POST /api/v1/setup/exchangeToken`
3. The API key is used for all subsequent authenticated requests

Setup tokens expire after 24 hours and can only be used once. Users can have multiple API keys for different clients/devices.

Events

`GET /api/v1/events`

- **Purpose:** Retrieve the authoritative event history.
- **Method:** GET
- **Request:**
  - No parameters
- **Response:**
  - Success (200 OK): A JSON array of event objects.
  - Unauthorized (401 Unauthorized): If the user is not authenticated.
- **Example Request:**

```
GET /api/v1/events
Authorization: Bearer <API_KEY>
```

- **Example Response:**

```
[
    {
        "uuid": "a1b2c3d4-e5f6-7890-1234-567890abcdef",
        "timestamp": 1678886400,
        "user": "user.123",
        "item": "task.456",
        "action": "create",
        "payload": "{}"
    },
    {
        "uuid": "b2c3d4e5-f6a7-8901-2345-67890abcdef0",
        "timestamp": 1678886401,
        "user": "user.123",
        "item": "task.456",
        "action": "update",
        "payload": "{\"title\": \"New Title\"}"
    }
]
```

`POST /api/v1/events`

- **Purpose:** Push new events from the client to the server.
- **Method:** POST
- **Request:**
  - A JSON array of event objects representing the new events.
- **Response:**
  - Success (200 OK): A JSON array of all event objects in the authoritative event history (after the new events have been applied and ACL validation).
  - Unauthorized (401 Unauthorized): If the user is not authenticated.
- **ACL Validation:** All incoming events are evaluated against current ACL. Events that violate the ACL are filtered out and not added to the history. See the ACL documentation for details.
- **Example Request:**

```
POST /api/v1/events
Authorization: Bearer <API_KEY>
Content-Type: application/json

[
    {
        "uuid": "c3d4e5f6-a7b8-9012-3456-7890abcdef01",
        "timestamp": 1678886402,
        "user": "user.123",
        "item": "item.789",
        "action": "create",
        "payload": "{}"
    }
]
```

- **Example Response:**

```
[
    {
        "uuid": "a1b2c3d4-e5f6-7890-1234-567890abcdef",
        "timestamp": 1678886400,
        "user": "user.123",
        "item": "item.456",
        "action": "create",
        "payload": "{}"
    },
    {
        "uuid": "b2c3d4e5-f6a7-8901-2345-67890abcdef0",
        "timestamp": 1678886401,
        "user": "user.123",
        "item": "item.456",
        "action": "update",
        "payload": "{\"title\": \"New Title\"}"
    },
    {
        "uuid": "c3d4e5f6-a7b8-9012-3456-7890abcdef01",
        "timestamp": 1678886402,
        "user": "user.123",
        "item": "item.789",
        "action": "create",
        "payload": "{}"
    }
]
```


Health Check

`GET /api/v1/health`

- **Purpose:** Check the health status of the service.
- **Method:** GET
- **Request:** None
- **Response:**
    - Success (200 OK): A JSON object containing the service health information.

- **Example Request:**

```
GET /api/v1/health
```

- **Example Response:**

```
{
    "status": "healthy",
    "timestamp": "2025-09-22T08:14:09Z",
    "version": "0.1.0",
    "uptime": 123
}
```

User Management

`POST /api/v1/user/resetKey`

- **Purpose:** Invalidate all API keys for a user, requiring them to re-authenticate.
- **Method:** POST
- **Authentication:** Required (API key)
- **Request:**
  - Query parameter: `user` (string) - ID of the user whose API keys to invalidate
- **Response:**
  - Success (200 OK): Confirmation message
  - Unauthorized (401): Insufficient permissions or invalid user
- **ACL:** Requires `.user.resetKey` permission for the target user, or `.root` access
- **Example Request:**

```
POST /api/v1/user/resetKey?user=user.123
Authorization: Bearer <ADMIN_API_KEY>
```

- **Example Response:**

```
{
    "message": "API keys invalidated successfully"
}
```

POST /api/v1/user/generateToken

- **Purpose:** Generate a setup token for a user.
- **Method:** POST
- **Authentication:** Required (API key)
- **Request:**
  - Query parameter: `user` (string) - ID of the user to generate setup token for
- **Response:**
  - Success (200 OK): Setup token information
  - Unauthorized (401): Insufficient permissions or invalid user
- **ACL:** Requires `.user.generateToken` permission for the target user, or `.root` access
- **Example Request:**

```
POST /api/v1/user/generateToken?user=user.123
Authorization: Bearer <ADMIN_API_KEY>
```

- **Example Response:**

```
{
    "token": "ABCD-1234",
    "expiresAt": "2025-09-26T12:00:00Z"
}
```

Setup

POST /api/v1/setup/exchangeToken

- **Purpose:** Exchange a setup token for an API key.
- **Method:** POST
- **Authentication:** None (token-based)
- **Request:**
  - JSON body with `token` (required) and optional `description`
- **Response:**
  - Success (200 OK): API key information
  - Unauthorized (401): Invalid, expired, or used token
- **Example Request:**

```
POST /api/v1/setup/exchangeToken
Content-Type: application/json

{
    "token": "ABCD-1234",
    "description": "Desktop Client"
}
```

- **Example Response:**

```
{
    "keyUuid": "550e8400-e29b-41d4-a716-446655440000",
    "apiKey": "sk_abcdefghijklmnopqrstuvwxyz1234567890",
    "user": "user.123",
    "description": "Desktop Client"
}
```

Access Control List (ACL)

The Access Control List (ACL) defines the relationships between users, items, and actions. It determines which users are allowed to perform which actions on which items.

Default Behavior

- All users can view all events. This means that Simple Sync is **only** appropriate for situations where **all** users of the system can be trusted to view **all** data in the system.
- By default, a user cannot perform any action on any item unless explicitly allowed by an ACL rule (deny all by default).
  - Note: there is a difference between viewing items and performing actions on items. All users can view all items because they can read all the events. However, they can not submit new events that perform actions on items without ACL rules to allow it.
- The `.root` user has implicit access to all items and actions, bypassing ACL checks.

ACL Structure

ACL rules are managed through events on a special `.acl` item. Each ACL event has an action of either `.acl.allow` or `.acl.deny` and a payload containing the rule details:

```
{
  "user": "string",
  "item": "string",
  "action": "string"
}
```

> **⚠ Caution**
>
> Rules with empty user, item, or action fields are not allowed.

Each field supports:

- Specific values (e.g., user ID, item ID, action name)

- Wildcard (`*`) for all matches
- Prefix wildcards (e.g., `admin.*`, `task.*`, `edit.*`) for prefix-based matching

ACL events are submitted via `POST /api/v1/events` and are validated against the current ACL before being added to the authoritative event history. Invalid ACL events are ignored.

Wildcard Support

The `user`, `item`, and `action` fields support the wildcard (`*`) to match all, and also support prefix-based wildcards (e.g., `task.*`, `admin.*`, `edit.*`) to match all that start with the specified prefix.

Rule Evaluation

ACL rules are evaluated based on specificity. For a given user, item, and action, the rule with the highest specificity score determines whether the action is allowed or denied. Specificity is calculated as the character count of the user, item, and action portions of a matching rule (wildcards are worth 0.5 specificity points).

1. Item specificity takes first precedence.
2. If there is a tie in item specificity, user specificity takes second precedence.
3. If there is a tie in user specificity, action specificity takes third precedence.
4. If there is still a tie, the most recently added rule (highest timestamp) takes precedence.

If no rule matches, the default behavior (deny all actions) applies.

Specificity Examples

For a request by user `user.123` to perform `edit` on item `task.456`:

Rule	Item	User	Action	Item Score	User Score	Action Score
A	*	*	*	0.5	0.5	0.5

Rule	Item	User	Action	Item Score	User Score	Action Score
B	*	user.123	*	0.5	8	0.5
C	task.*	*	*	5.5	0.5	0.5
D	*	*	edit	0.5	0.5	4

Compare item specificity: Rule C (5.5) > others (0.5) → Rule C wins.

For item `task.456` and action `edit`, assuming no higher item matches:

Rule	Item	User	Action	Item Score	User Score	Action Score
E	task.*	*	edit	5.5	0.5	4
F	*	*	edit	0.5	0.5	4

Item specificity: Rule E (5.5) > Rule F (0.5) → Rule E wins.

For item `task.456`, user `admin.123`, action `edit`, with item specificity tied:

Rule	Item	User	Action	Item Score	User Score	Action Score
G	task.*	*	*	5.5	0.5	0.5
H	task.*	admin.*	*	5.5	6.5	0.5

Item specificity tied (5.5), compare user: Rule H (6.5) > Rule G (0.5) → Rule H wins.

For item `task.456`, user `admin.123`, action `edit.description`, with item and user specificity tied:

Rule	Item	User	Action	Item Score	User Score	Action Score
I	task.*	admin.*	*	5.5	6.5	0.5
J	task.*	admin.*	edit.*	5.5	6.5	5.5

Item and user tied, compare action: Rule J (5.5) > Rule I (0.5) → Rule J wins.

Rule Examples

To allow all users to mark “task.123” as complete:


```
{
  "uuid": "01997af2-df11-73b3-8329-e5c3affc9a05",
  "timestamp": 1758704361233,
  "user": "admin.user1",
  "item": ".acl",
  "action": ".acl.allow",
  "payload": "{\"user\": \"*\", \"item\": \"task.123\", \"action\":
\"markComplete\"}"
}
```


To allow user “user.456” to edit any item:


```
{
  "uuid": "01997af3-4299-7be7-8bd7-d01636e06d73",
  "timestamp": 1758704386713,
  "user": "admin.user1",
  "item": ".acl",
  "action": ".acl.allow",
  "payload": "{\"user\": \"user.456\", \"item\": \"*\", \"action\": \"edit\"}"
}
```


To allow all admin users to perform any delete action on any task:


```
{
  "uuid": "01997af3-7a2f-7b65-9055-8439f87d7450",
  "timestamp": 1758704400943,
  "user": "admin.user1",
  "item": ".acl",
  "action": ".acl.allow",
  "payload": "{\"user\": \"admin.*\", \"item\": \"task.*\", \"action\":
\"delete.*\"}"
}
```

ACL Management

ACL rules are managed by submitting events to the `POST /api/v1/events` endpoint with `item` set to `.acl` and `action` set to `.acl.allow` or `.acl.deny`. The payload must contain `user`, `item`, and `action` fields defining the rule. ACL events are validated against the current ACL before being added to the authoritative history; invalid ACL events are ignored.

The current ACL state can be inferred from the authoritative event history by filtering for `.acl` events. See the v1 API Specification for details on event submission.

Note: ACL events require appropriate permissions based on existing rules. The `.root` user has implicit access to submit any ACL events.

Internal Events

Any event that uses a `.` prefix for the item or action is **reserved** for internal usage.

Internal events are used in two ways.

First, to allow users to trigger special functionality within the server. These events are marked **Trigger: User** below.

Second, to allow the server to create an audit history for all actions triggered through the API. These events are marked **Trigger: API** below. These events will **always** be rejected if users attempt to add them.

ACL

Trigger: User

The `.acl` item is used for updating the ACL. The payload must contain a valid ACL rule. The event's `action` field must be either `.acl.allow` or `.acl.deny`.

Users

The `.user.` item prefix is used for events related to user administration and authentication.

Create User

Trigger: User

The `.user.create` action is used for creating new users. The new user's ID is given in the event's `item` field (for example `"item": ".user.bob"`). If the given user ID already exists, the event is rejected.

Generate User Token

Trigger: API

The `.user.generateToken` action is used to log calls to the `/api/v1/user/generateToken` API endpoint.

Exchange User Token

Trigger: API

The `.user.exchangeToken` action is used to log calls to the `/api/v1/user/exchangeToken` API endpoint.

Reset User Key

Trigger: API

The `.user.resetKey` action is used to log calls to the `/api/v1/user/resetKey` API endpoint.