Overview

What is Procivis One?

Procivis One is a self-hosted platform for digital credentials and identity wallets.

Issue verifiable credentials, verify presentations, and manage digital identities using open standards. The platform runs in your infrastructure with the open source Core component and optional enterprise components.

On this page

What do you want to do?

🚀 Get Started Quickly

New to Procivis One? Start here:

- Run the Core locally - Set up the open source Core in minutes
- Deploy the full stack - Enterprise stack with Docker Compose
- Try the Desk UI - Request access

to hosted trial environment

💻 Build with APIs

Integrate credential workflows into your application:

- Issue your first credential - Tutorial: Create and issue a credential
- Verify your first credential - Tutorial: Request and verify a presentation
- Core API reference - Complete API documentation

🎨 Use the Desk UI

Manage credentials through the web interface:

- Desk UI overview - Introduction to the no-code interface
- Getting Started Tutorial - Full lifecycle tutorial
- Country Profiles - Understand how country profiles make interop easy

📚 Learn More

Understand the platform and standards:

- Components overview - Server

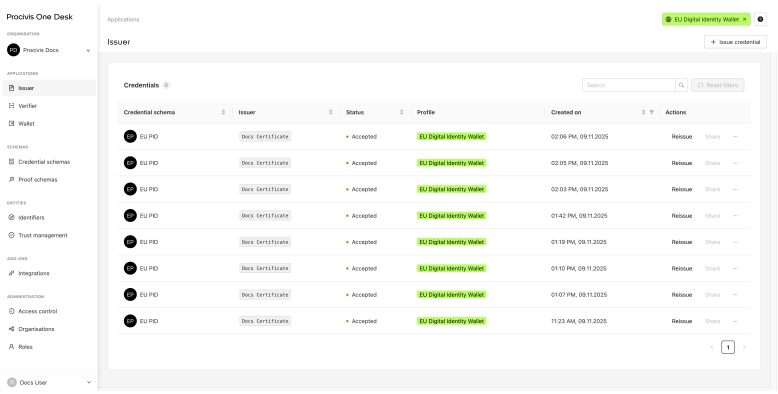
and mobile components ready to deploy

- Supported standards - OpenID4VC, W3C VCs, ISO mdocs, eIDAS 2.0
- Implementation approaches - Choose your deployment path

Quick access: API Reference • SDK Documentation

Overview

Desk UI



Credentials issued using the EUDI Wallet profile

On this page

Procivis One Desk UI is a user interface for managing the complete lifecycle of credentials. It supports many workflows including:

- Issuing and verifying credentials
- Holding credentials in an organizational wallet
- Creating and managing cryptographic keys, DIDs, and X.509 certificates
- Managing wallet unit attestations as a wallet provider
- System administration including roles, permissions, and organizational management

This section provides workflow guides for using the Desk.

Who should use this section

Trial users: Evaluating Procivis One through our hosted trial environment

Desk users: Operating a self-deployed instance with the Desk frontend

> NOTE
>
> The trial includes most common features. Some advanced features described here may only be available in self-deployed instances. Self-deployed instances may also have limited features depending on configuration.

Get started with the trial

To access our trial deployment, visit the Trial access page.

Download the wallet:

NOTE

The publicly available Wallet app connects to the trial deployment. Some features only work with the trial environment.

Download the verifier app:

NOTE

The publicly available Verifier app can only be initialized from the trial deployment of the Desk.

Learning the Desk

New to Procivis One Desk? Start with the Getting started tutorials which walk through the complete lifecycle of issuing, holding, and verifying credentials.

Need help with a specific task? Browse the workflow tutorials or use the in-app help dropdown in the Desk for context-relevant documentation links.

Overview

APIs

Core → Download OpenAPI specification

Enterprise Backend → Download OpenAPI specification

OpenID Bridge → Download OpenAPI specification

Authentication and authorization

Core

The Core API's authentication requirements depend on the configured authentication mode. For detailed information about authentication architecture and configuration, see Authentication and authorization.

Authentication by mode

No authentication mode

When Core is configured with `INSECURE_NONE`, all API endpoints are accessible without authentication. No `Authorization` header is required.

Example request:

```
curl -L '/api/organisation/v1' \
  -H 'Accept: application/json'
```

> ⚠ WARNING
>
> This mode should only be used in development environments or when authentication is enforced by infrastructure components. See Authentication and authorization for security considerations.

Static token mode

When Core is configured with `STATIC` mode, include the configured static token in the `Authorization` header as a Bearer token.

Example request:

```
curl -L '/api/organisation/v1' \
  -H 'Accept: application/json' \
  -H 'Authorization: Bearer your-static-token'
```

> ⚠ WARNING
>
> This mode should only be used in development environments or when authentication is enforced by infrastructure components. See Authentication and authorization for security considerations.

STS mode

When Core is configured with `STS` mode, include a valid STS application token in the `Authorization` header as a Bearer token.

Example request:

```
curl -L '/api/organisation/v1' \
```

```
  -H 'Accept: application/json' \
  -H 'Authorization: Bearer
eyJhbGciOiJFZERTQSIsImtpZCI6IjIwMjQtMDktZWQyN
```

Token requirements:

- Must be a valid JWT signed with keys published at the configured JWKS endpoint
- Must contain valid `aud`, `iss`, and `exp` claims matching Core's configuration
- Must include the the `organisationId` (tenant) for the resources being accessed
- Must include the `permissions` array with the necessary permissions for the requested operation

Obtaining STS tokens:

For information on how to obtain STS tokens, see Obtaining STS tokens.

Authorization failures

If the token in invalid, expired, or lacks required permissions, the API returns a `401 Unauthorized` or `403 Forbidden` response.

API surface overview

The Core API consists of three categories of endpoints with different authentication requirements.

Management endpoints: `/api/*`

Management endpoints provide administrative and operational functionality for credentials, schemas, organizations (tenants), and other platform resources.

These endpoints respect the configured authentication mode:

- No authentication mode: All endpoints are accessible without credentials
- Static token mode: Requires the configured static token in the `Authorization` header
- STS mode: Requires a valid STS application token with appropriate permissions

Most operations require specific permissions (for example, `CREDENTIAL_ISSUE`) which are checked when using STS mode.

STS endpoints: `/ssi/*`

SSI (Self-Sovereign Identity) endpoints implement standard protocols for credential issuance and verification, including OpenID4VC.

Many SSI endpoints are public and do not require authentication, as they implement protocol-specific security mechanisms. Some endpoints require protocol-specific tokens, for example:

- `/ssi/openid4vci/draft-13/{id}/credential` requires an OpenID4VC access token obtained through the credential offer flow
- `/ssi/trust-entity/v1/*` endpoints require a remote-agent token for trust registry operations

These protocol-specific tokens are independent of the Core's authentication mode and are passed in request headers as defined by their respective specifications.

System endpoints

System endpoints provide service health and operational information:

- /build-info - Build and version details

- /metrics - System metrics
- /health - System health check

When enabled in the configuration, these endpoints are public and accessible regardless of authentication mode.

Enterprise Backend

The Enterprise Backend API (or "Desk API") authenticates requests using IAM-issued bearer tokens. Include a valid IAM access token in the `Authorization` header with each request.

Authentication flow

1. **Authenticate with IAM**: Obtain an access token from your configured IAM provider (for example, Okta, Auth0, Keycloak)

2. **Call Desk endpoints**: Include the IAM access token as a bearer token in the `Authorization` header

3. **Automatic token exchange**: The service validates your IAM token and exchanges it internally for STS application tokens when calling downstream services

Making authenticated requests

Include the session cookie in your requests:

```
curl -L '/api/organisation/v1' \
  -H 'Accept: application/json' \
  -H 'Authorization: Bearer your-access-
```

```
token'
```

IAM token requirements

Your IAM provider must issue tokens that conform to the expected structure. See IAM token structure for details on required claims and configuration.

Downstream service authentication

The `one-frontend-backend` handles downstream authentication automatically. When you call an endpoint with your IAM token, the service:

1. Validates your IAM access token
2. Exchanges it for an STS application token using the internal Secure Token Service
3. Forwards requests to Core or other services with the appropriate authentication

As an API client, you do not need to manage downstream tokens. The service's authentication mode for calling downstream services is configured at deployment.

Common patterns

Errors

HTML status code responses follow the common

pattern:

- 2xx: Successful responses
- 4xx: Client error responses
- 5xx: Server error responses

Most error responses offer additional information for troubleshooting.

Pagination

Pagination is handled similarly across different resources.

Query parameters

When retrieving lists of objects (keys, DIDs, credential schemas, and others), optionally pass two different query parameters to control what is returned and how: `page` and `pageSize`.

- Passing a `page` value returns a specific page of the results. The first page is `0`.
- Passing a `pageSize` value controls how many items appear on each page. The default value for the Desk API is `30`. For the Core API, a value must be specified.

Returned values

When lists are returned, there are two pagination-related values returned: `totalItems` and `totalPages`.

- The `totalItems` value is the total number of the associated items.
- The `totalPages` value is the total number of pages the list has.

Sorting

Sorting is handled similarly across different resources. Pass query parameters to sort lists.

There are two sorting values:

- `sort`: sort results by a result in the response value. Not all response values are supported. Supported values are listed in each endpoint reference description.
- `sortDirection`: order the results by increasing (`ASC`) or decreasing values (`DESC`). Supported values:
  - `ASC`
  - `DESC`

If no sorting values are provided, `createdDate` + `DESC` are used.

If a value is passed for `sort`, the default direction becomes `ASC`.

Filtering

Filtering is handled consistently across different resources. Pass query parameters to filter lists and find specific items efficiently.

The following resources support filterable lists:

- Keys
- Identifers
- Credential schemas
- Proof schemas
- Credentials
- Proof requests

- Trust anchors
- Trust entities
- Wallet units

Below are the most important filtering capabilities available across these resources. Note that not every filtering method is available for all resource types; each endpoint supports a subset of these options based on the nature of the data being filtered.

By text

Name

The `name` parameter filters items whose names start with the provided string. The search is case-insensitive, so "digi" will match items starting with "Digital", "DIGITAL", or "digital".

Credentials and proof requests do not have names; for these resources this parameter filters by the schema used. For example, "U.S." will match credentials whose schema starts with "U.S. Passport" or "U.S. Driver License".

Exact matching

The `exact[]` parameter modifies how text filters behave, changing them from "starts with" matching to exact string matching, still case-insensitive. The parameter accepts multiple values using either array notation (`exact[]=name&exact[]=schemaId`) or repeated parameters (`exact=name&exact=schemaId`).

For example, without `exact[]`, searching for "Digital" would match both "Digital" and "Digital Passport Schema". With `exact[]=name`, only items named exactly "Digital" would be returned.

Advanced search

The `/credential` and `/history` resources support advanced search capabilities using the `searchText` parameter combined with `searchType[]` to specify where to search.

The `searchText` parameter accepts a search string, while `searchType[]` determines which fields to search within. Unlike the name parameter which only matches items that start with the provided string, `searchText` searches for the string anywhere within the specified fields.

This advanced search is particularly useful for finding credentials based on their content rather than just metadata. For example, using "John" as the `searchText` with CLAIM_VALUE as the `searchType[]` will return any credentials that contain "John" anywhere within their claim values. If no `searchType` is specified, all fields in the enum are searched.

By date range

Creation and modification dates

All list endpoints support filtering by when items were created or last modified using date range parameters. These filters accept timestamps in RFC3339 format (e.g., "2023-06-09T14:19:57.000Z").

Use `createdDateAfter` and `createdDateBefore` to filter by creation time, or `lastModifiedAfter` and `lastModifiedBefore` to filter by modification time. You can use these parameters individually or combine them to create specific date ranges.

For example, to find all credential schemas created in

the last 30 days, use `createdDateAfter` with a timestamp from 30 days ago. To find items modified within a specific week, combine `lastModifiedAfter` and `lastModifiedBefore` with the appropriate start and end timestamps.

Entity-specific dates

Some resources offer additional date filters specific to their lifecycle events. These use the same RFC3339 timestamp format as creation and modification dates.

Credentials support `issuanceDateAfter` and `issuanceDateBefore` to filter by when credentials were actually issued to holders, as well as `revocationDateAfter` and `revocationDateBefore` for finding revoked credentials within specific timeframes.

Proof requests provide `requestedDateAfter` and `requestedDateBefore` to filter by when the proof request was sent to the holder, and `completedDateAfter` and `completedDateBefore` to find proofs that were successfully submitted within a given period.

By entity states

Status/state filtering

Many resources allow filtering by their current state or status within the system. The available states vary by resource type, reflecting each entity's specific lifecycle.

Use the `states[]` array to filter by one or more states.

Related state guides:

- Issuing
- Verifying
- Holding

Role-based filtering

The same instance of the system can be used for any combination of issuing, holding, and verifying. Use the `roles[]` array to filter by one or more roles of the system.

For example, if the system is used to issue credentials and also as an organizational wallet, it will contain credentials it has issued and credentials it holds as a wallet. Filter by `HOLDER` to return credentials the system holds as a wallet.

Similarly, if the system is used to verify credentials and also as an organizational wallet, it will contain proofs it requested and proofs it shared with some other verifier. Filter by `VERIFIER` to return proofs the system requested from a wallet holder.

Active vs. deactivated entities

Some resources allow filtering by whether entities are currently active or have been deactivated.

Identifiers support filtering by operational status using the `state` parameter with values `ACTIVE` or `DEACTIVATED` to find only functional identifiers or those that have been taken out of service.

By UUID

Most list endpoints support filtering by specific entity identifiers when you need to retrieve particular items. This is useful for targeted queries or when working with known sets of entities.

Use the `ids[]` parameter to filter by specific UUIDs. This accepts an array of identifiers, allowing you to retrieve multiple specific items in a single request rather than making individual calls for each entity.

By technical specifications

Formats and types

Many resources support filtering by their technical format or type, allowing you to work with specific implementations or standards.

Credential schemas can be filtered using `formats[]` to find schemas that use particular credential formats like `SD_JWT_VC` or `MDOC`. Note that you should enter a string matching the desired format as it is named in your configuration.

Identifiers support filtering by `types[]` with values like `DID`, `KEY`, or `CERTIFICATE` to distinguish between different identifier mechanisms. You can also filter by `didMethods[]` to find identifiers using specific DID methods like `WEB` or `KEY`.

Trust entities and trust anchors provide `type[]` filtering to distinguish between different trust management approaches and entity types within your trust infrastructure.

Key algorithms and storage

Resources that work with cryptographic keys support filtering by their technical specifications, helping you manage keys based on their security properties and storage requirements.

Use `keyTypes[]` on the keys endpoint to filter by

cryptographic algorithms like EDDSA or ECDSA. For identifiers, use `keyAlgorithms[]` to find identifiers that either use a specific algorithm directly (for key-based identifiers) or are associated with keys using that algorithm (for DIDs and certificates).

The `keyStorages[]` parameter filters by where keys are stored, with options like `INTERNAL`, `HARDWARE`, or other storage types defined in your system configuration.

By relationships

Remote vs. local entities

Many resources distinguish between entities created locally within your system versus those encountered through interactions with external actors. This helps separate your own managed entities from those belonging to external parties.

Use the `isRemote` parameter with boolean values to filter identifiers and keys. Setting `isRemote=false` returns only entities you created and control, while `isRemote=true` returns entities from external wallets, issuers, or verifiers that your system has interacted with.

Schema associations

Resources often support filtering by their associated schemas, helping you find entities related to specific credential or proof templates.

Proof requests support filtering by `proofSchemaIds[]` to find all verification interactions that used specific proof schemas.

Trust anchor relationships

Trust-related resources support filtering by their relationships within trust management hierarchies, helping you organize and query trust infrastructure.

Trust entities can be filtered using `trustAnchorId` to find all entities associated with a specific trust anchor, making it easy to see which issuers or verifiers belong to a particular trust framework.

Trust anchors themselves support filtering by `isPublisher` with values true or false to distinguish between trust anchors you publish versus those you subscribe to from external sources.

Access & Licensing

Open Source: Core API is available under open source license
→ Procivis One Core on GitHub

Enterprise: Desk API and OpenID Bridge require an enterprise license
→ Contact sales

On this page