Implementing a Nuts v2 Use Case

A Nuts use case enables organizations that don't directly trust each other, but rely on trusted third parties, to find each other and securely exchange data. The use case specifies which data is exchanged, on which authorization grounds, and how API endpoints can be found.

This book tells you how to implement a use case that uses Nuts spec v2: deployment and configuration, managing DIDs and credentials, registering on and searching the Discovery Service, and requesting and validating access tokens. Nuts spec v2 uses did:web and the Discovery Service. It does NOT use did:nuts or the gRPC network.

This book uses **360-graden-clientbeeld** as its example use case. Steps that are the same for every use case are written in general terms. Everything specific to 360-graden-clientbeeld is in a blue box marked "360-graden-clientbeeld" (see "How to use this book" for an example). For another use case, refer to that use case's implementation guide (IG) instead.

- Start here
  - How to use this book
- Deploy
  - Deploy the Nuts node
  - Endpoints and TLS certificates
  - Deploy admin app
  - Configure url rewrite for OAuth
- Configure the node for a use case
  - Install the use case's configuration files
- Set up a care organization
  - Create a subject

    - Get an X509Credential from a UZI certificate
    - Load the credential into the wallet
    - Obtaining credentials

- Join the use case
    - Register on the Discovery Service

- Outbound: querying another party
    - Find care organizations and their endpoints
    - Request an access token
    - Make the FHIR call

- Inbound: handling queries
    - Validate the access token

- Test your setup
    - Test your setup step by step

- Running it
    - Renew credentials
    - Operate the node

Start here

How this book works, who does what, and what you need from the use case.

How to use this book

This book takes you through implementing a Nuts spec v2 use case, from deploying the Nuts node to querying another care organization. Each page is one task: what it's for, what you need first, the steps, and how to check it worked.

Everything specific to the example use case, 360-graden-clientbeeld, is in a blue box like this one:

> **360-graden-clientbeeld**
>
> Values and rules that only apply to 360-graden-clientbeeld. For another use case, refer to that use case's implementation guide (IG).

Who does what

Party	Role
Vendor	Runs the Nuts node, for all of its care organizations, and integrates it with its EHR or other software. Most of this book is for the vendor.
Care organization	Owns its UZI server certificate and creates its X509Credential with it (see "Get an X509Credential from a UZI certificate").
Use case	Its implementation guide (IG) sets the rules: which credentials, which endpoints, which data, and on which legal basis.

What you need from the use case's IG

What	Used in
Discovery Service definition, per environment	"Install the use case's configuration files"
Policy definition, per environment	"Install the use case's configuration files"
Which credentials a care organization needs	"Set up a care organization"
Which registration parameters to publish, e.g. API endpoints	"Register on the Discovery Service"
The scope, and which credentials go in a token request	"Request an access token"

What	Used in
Which queries are allowed, and the rules for allowing them	"Make the FHIR call", "Validate the access token"

Checklist

Step	How often	Chapter
Deploy the Nuts node	Once per vendor, per environment	Deploy
Install the use case's configuration files	Once per use case, per environment	Configure the node for a use case
Create a subject, get and load its credentials	Once per care organization, per environment	Set up a care organization
Register on the Discovery Service	Once per care organization, per use case, per environment	Join the use case
Find the other party, request a token, make the call	Every query	Outbound: querying another party
Validate the token, decide whether to allow the query	Every incoming query	Inbound: handling queries

Words used in this book

Word	Meaning
Subject	A care organization on your Nuts node. You use its subject ID in all calls to the node's internal API.
DID	Decentralized identifier: a public ID with keys. Each subject has one or more DIDs.
did:web	The DID method used in spec v2. Its DID document is published on your node's public web address.
Verifiable Credential	A signed statement about a subject, e.g. its URA number (X509Credential). Credential types are defined in the credential catalog.
Verifiable Presentation	One or more credentials, signed by the holder, to show them to another party.
Wallet	Where the node stores a subject's credentials.
Discovery Service	A shared list where care organizations register, so others can find them and their endpoints.
Discovery Server	The server that holds a Discovery Service's list. The use case says where it is.
Policy definition	The credentials another party must present to get an access token, per scope.

Word	Meaning
Scope	The name of what an access token gives access to. The use case sets it.
Authorization server	The part of a Nuts node that issues access tokens.
Introspect	Ask your Nuts node what an access token contains and whether it's valid.
DPoP	A way to bind an access token to the requester's key, so a stolen token can't be used.

Deploy

Deploy the Nuts node and, optionally, Nuts Admin. Put the node behind a reverse proxy with the right TLS certificates, and check that it's reachable.

Deploy the Nuts node

There are two ways to run the Nuts part of your system. Choose one:

Option	What it is	When to choose it
Nuts node (default)	The Nuts node on its own. You build the integration with your EHR yourself, following this book.	You want full control, or you already integrate with the Nuts node
Nuts Knooppunt	Has an embedded Nuts node, plus components for integrating with the MinVWS Generic Functions (Generieke Functies), such as a policy decision point (PDP) for access decisions and Mitz integration	You want more ready-made parts, or you integrate with the Generic Functions. See the Knooppunt section below

One Nuts node per vendor (and per environment) hosts all of the vendor's care organizations.

Nuts node

The Nuts node documentation describes deployment in detail. Start with the recommended deployment and running on Docker.

Install

The Nuts node is published as:

Form	Where
Docker image	`nutsfoundation/nuts-node` on Docker Hub
Helm chart, for Kubernetes	Helm repository `https://nuts-foundation.github.io/nuts-node`, chart `nuts-node-chart` (source)
Binaries (Linux, amd64 and arm64)	GitHub releases

Use a fixed version (e.g. `nutsfoundation/nuts-node:<version>`), not `latest`. For the latest version, see the GitHub releases. Docker images are signed; see verifying image signatures.

Configuration

The settings you need for a spec v2 use case. Each can be set in `nuts.yaml` or as an environment variable (`NUTS_` + the name in capitals, with `_` instead of `.`, e.g. `NUTS_HTTP_INTERNAL_ADDRESS`).

Setting	Value	Why
`url`	Your node's public URL, e.g. `https://nuts.example.com`	Used in your did:web DIDs and OAuth2 URLs. Must be HTTPS.
`strictmode`	`true` (default)	Forbids insecure settings. Only turn it off on development or test systems without production data.
`didmethods`	`web`	Spec v2 only uses did:web (see "Create a subject").
`http.internal.address`	e.g. `:8081`	The internal API. Default `127.0.0.1:8081`, which can't be reached from outside the container.
`storage.sql.connection`	Connection string to PostgreSQL, MySQL or SQL Server (including Azure SQL)	The default, SQLite, isn't meant for production.
`crypto.storage`	`vaultkv` (HashiCorp Vault) or `azure-keyvault`	Where private keys are stored. `fs` (on disk) is only for development.
`discovery.definitions.directory`	Directory with the Discovery Service definitions	See chapter "Configure the node for a use case".
`policy.directory`	Directory with the policy definitions	See chapter "Configure the node for a use case".

Network

- Put a reverse proxy in front of the node that terminates TLS. See "Endpoints and TLS certificates" for which endpoints to expose, and with which certificate.
- Never expose the internal API (`:8081`) to the internet.
- If the node runs on a subpath (e.g. `https://example.com/nuts`), rewrite the `.well-known` URLs. See "Configure url rewrite for OAuth".

Check that it works

- `GET <internal Nuts interface>/health` returns `200 OK` with status `UP`.
- After creating your first subject, check that its DID resolves from the internet (see "Create a subject").

Nuts Knooppunt

Knooppunt has an embedded Nuts node, so the steps in this book still apply. It also has a policy decision point (PDP) that helps you decide whether to allow an inbound query (see "Validate the access token").

See the Knooppunt documentation, in particular:

- Deployment

- Configuration
- Integration, including the PDP
- An example docker-compose.yml

Endpoints and TLS certificates

Which endpoints you expose, to whom, and with which TLS certificate. This repeats the recommended deployment for spec v2 use cases; see there for details.

Nuts node endpoints

Your reverse proxy terminates TLS for the public endpoints.

Endpoint	Interface	Who uses it	Protection
`/iam`	Public (`:8080`)	Other parties resolving your did:web DID documents	HTTPS with a **publicly trusted** server certificate
`/.well-known`	Public (`:8080`)	Other parties: DID documents, OAuth2 metadata	HTTPS with a **publicly trusted** server certificate
`/oauth2`	Public (`:8080`)	Other Nuts nodes requesting access tokens from you	HTTPS with a **publicly trusted** server certificate
`/statuslist`	Public (`:8080`)	Parties checking whether credentials you issued are revoked	HTTPS with a **publicly trusted** server certificate
`/discovery`	Public (`:8080`)	Only if your node runs a Discovery Service server	HTTPS with a **publicly trusted** server certificate
`/internal`	Internal (`:8081`)	Your own applications, Nuts Admin	Never public: restrict access through network separation
`/status`, `/status/diagnostics`, `/health`, `/metrics`	Internal (`:8081`)	Monitoring, health checks	Restrict access through network separation
`/n2n`, `/public`, gRPC (`:5555`)	—	Spec v1 (did:nuts) only	Don't expose them

Your data endpoints

Protect your FHIR API (or other data API) with mutual TLS (mTLS) based on PKIoverheid certificates: both your server and the calling vendor's client authenticate with a PKIoverheid certificate. Check the use case's IG for what's allowed in test environments.

Don't mix up the certificates

Certificate	Used for
UZI server certificate	Creating the care organization's X509Credential (see "Get an X509Credential from a UZI certificate")
Publicly trusted TLS certificate	The Nuts node's public endpoints
PKIoverheid certificate	mTLS on your data endpoints

Deploy admin app

Nuts Admin is an optional web application for managing your Nuts node by hand: create subjects, load and issue credentials, and register on Discovery Services. It's useful for testing and for small numbers of care organizations. For many care organizations, integrate with the node's internal API instead, as described in the rest of this book.

Run it

```
docker run -p 1305:1305 -e NUTS_NODE_ADDRESS=http://nutsnode:8081 nutsfoundation/nuts-admin:<version>
```

- `NUTS_NODE_ADDRESS` points to the node's **internal** API.
- Nuts Admin listens on port `1305` (change it with `PORT`).
- For the latest version, see the tags on Docker Hub.

See the Nuts Admin README for all settings.

Secure it

Nuts Admin passes requests on to the node's internal API. Anyone who can reach Nuts Admin can manage your node.

- Turn on user login with OpenID Connect (`NUTS_OIDC_ENABLED=true`, plus the settings in the README), or
- restrict access to Nuts Admin in another way, e.g. network separation.

Never expose Nuts Admin to the internet without login.

Configure url rewrite for OAuth

Other parties look up your node's OAuth2 metadata at a `.well-known` URL, which always starts at the root of the domain (RFC 8615). If your node runs at the root of its own (sub)domain, e.g. `https://nuts.example.com`, you don't need to do anything.

If the node runs on a subpath, e.g. `https://example.com/nuts`, your reverse proxy must forward the OAuth2 metadata URL to the node. Example for a node on `/nuts`:

Other parties request	Forward to the node as
`https://example.com/.well-known/oauth-authorization-server/nuts/oauth2/<subject ID>`	`/.well-known/oauth-authorization-server/oauth2/<subject ID>`

In short: remove the subpath (`/nuts`) after the `.well-known/...` part.

The node serves all its other endpoints at the root too (`/iam`, `/oauth2`, `/statuslist`, ...). So your proxy must also remove `/nuts` from those, e.g. forward `https://example.com/nuts/oauth2/...` to `/oauth2/...`.

NGINX

```
location /.well-known/oauth-authorization-server/nuts/ {
    proxy_pass http://nuts-node:8080/.well-known/oauth-authorization-server/;
}
```

Azure Application Gateway

```
path: '/.well-known/oauth-authorization-server/nuts'
rewriteCondition: '/.well-known/oauth-authorization-server/nuts(.*)'
rewriteRule: '/.well-known/oauth-authorization-server{var_uri_path_1}'
```

Check that it works

From outside your network, open:

```
https://example.com/.well-known/oauth-authorization-server/nuts/oauth2/<subject ID>
```

It should return JSON with your node's OAuth2 metadata, e.g. an `issuer` of `https://example.com/nuts/oauth2/<subject ID>`.

Configure the node for a use case

Install the Discovery Service and policy definitions you receive from the use case.

Install the use case's configuration files

A use case gives you two kinds of configuration files. Your Nuts node needs both, for every use case you join.

File	What it tells your node	Directory setting
Discovery Service definition	Where the use case's Discovery Service is, and which credentials a care organization must present to register on it	`discovery.definitions.directory`
Policy definition	Which credentials another party must present to get an access token from you, per scope	`policy.directory`

Do this once per use case, per environment.

Don't change these files. They contain the use case's trust rules, such as which certificate authorities are trusted. A changed file can make your node accept parties it shouldn't.

Install the files

1. Download the Discovery Service definition and the policy definition for your environment from the use case's IG.
2. Put them in the directories set in `discovery.definitions.directory` and `policy.directory`. Use any file name that ends in `.json`; the node skips other files.
3. If you use Docker, mount the directories read-only.
4. Restart the node. It reads the files at startup. If it can't read a Discovery Service definition, it doesn't start; the log says which file.

Each environment has its own files. Only install the files for the environment the node runs in. A use case typically uses the same service ID and scope in all environments; with files for two environments, the node won't start.

360-graden-clientbeeld

The files are in the nl-360-ig repository:

Environment	Discovery Service definition	Policy definition	Discovery Server

Development	discovery_360_dev.json	policy_360_dev.json	`http://localhost:9091` (a local Discovery Server)
Test	discovery_360_test.json	policy_360_test.json	`https://development.nedap-nuts.nl`
Acceptance	discovery_360_acc.json	policy_360_acc.json	`https://staging.nedap-nuts.nl`
Production	discovery_360_prod.json	policy_360_prod.json	`https://nedap-nuts.nl`

All environments use the service ID `360-graden`.

What's in a Discovery Service definition

Part	What it means
`id`	The service ID. You use it in the Discovery Service API calls.
`endpoint`	The Discovery Server where registrations are stored
`presentation_max_validity`	How long a registration is valid, in seconds. Your node refreshes it before then.
`presentation_definition.input_descriptors`	The credentials a care organization must present to register
Fields with a `filter`	Requirements on a credential, e.g. which certificate authorities are trusted. This is why each environment has its own file.
Fields with an `id`	Values the node takes from the credentials. They show up as `fields` in search results. A pattern with a capture group, like `(\d+)`, returns only the captured part.

360-graden-clientbeeld

The test definition, shortened: some fields, like `organization_id`, `organization_city` and `fhir_base_url_stu3`, are left out. It requires an X509Credential from a trusted certificate authority, and a DiscoveryRegistrationCredential with the endpoints. The `organization_ura` pattern returns only the URA.

```
{
  "id": "360-graden",
  "endpoint": "https://development.nedap-nuts.nl/discovery/360-graden",
  "presentation_max_validity": 259200,
  "presentation_definition": {
```

```
    "input_descriptors": [
      {
        "id": "id_uzicert_x509credential",
        "constraints": {
          "fields": [
            { "path": ["$.type"], "filter": { "const": "X509Credential" } },
            { "path": ["$.issuer"], "filter": { "pattern":
"^did:x509:0:sha256:(GwlhBZ...|szqMaT...|svKzNx...)..." } },
            { "id": "organization_ura", "path": ["$.credentialSubject.san.otherName"],
"filter": { "pattern": "^[0-9.]+-\\d+-\\d+-S-(\\d+)-00\\.000-\\d+$" } },
            { "id": "organization_name", "path": ["$.credentialSubject.subject.O"] }
          ]
        }
      },
      {
        "id": "id_registration_parameters_credential",
        "constraints": {
          "fields": [
            { "path": ["$.type"], "filter": { "const": "DiscoveryRegistrationCredential"
} },
            { "id": "authorization_server_url", "path":
["$.credentialSubject.authServerURL"] },
            { "id": "fhir_base_url_r4", "path": ["$.credentialSubject.fhirR4BaseURL"],
"optional": true }
          ]
        }
      }
    ]
  }
}
```

What's in a policy definition

A policy definition maps a scope to the credentials another party must present to get an access token for it.

- The top-level key is the scope. Another party asks for it in its token request.
- Under the scope, `organization` lists the credentials the requesting care organization must present.

- The fields with an `id` show up in the token introspection result (see "Validate the access token").

360-graden-clientbeeld

The policy, shortened. The scope is `360-graden`. The requester must present an X509Credential and a NutsEmployeeCredential.

```
{
  "360-graden": {
    "organization": {
      "input_descriptors": [
        { "id": "id_uzicert_uracredential", "...": "X509Credential from a trusted CA, as
in the Discovery Service definition" },
        { "id": "id_employee", "...": "NutsEmployeeCredential with employee_name,
employee_role and employee_identifier" }
      ]
    }
  }
}
```

Set up a care organization

Do this once per care organization, on the vendor's Nuts node: create a subject, get an X509Credential from the organization's UZI server certificate, and load it into the wallet.

Create a subject

Each care organization gets its own subject on the vendor's Nuts node. A subject is the name you use for the care organization in all calls to the Nuts node's internal API. Behind a subject, the node creates one or more DIDs (decentralized identifiers) and their keys.

Do this once per care organization, per environment.

Before you start

- The Nuts node is deployed (see chapter "Deploy").
- did:web is enabled by default. We recommend setting `didmethods` to `web`, so the node doesn't also create a did:nuts DID, which spec v2 use cases don't need.

Decide: generated or your own subject ID

You have to choose one of these:

Option	How	What you do
Let the Nuts node generate the subject ID	Leave out `subject` in the request	Store the subject ID from the response in your own system, with the care organization
Bring your own subject ID	Set `subject` in the request, e.g. to the ID your system already uses for the care organization	Nothing to store: you already know the ID

Keep in mind:

- The subject ID is public. It's part of your authorization server URL (`https://<your node>/oauth2/<subject ID>`), which is published on the Discovery Service and is in every access token. Don't use an ID you want to keep private.
- It may only contain letters, digits, `.`, `_` and `-`.

If you bring your own subject ID and it isn't random (e.g. a KVK number or an auto-incrementing database key), consider adding the environment at the end, e.g. `12345678-acc`. The same ID then can't exist in two environments, which reduces the chance of leaking data because environments got mixed up. The examples on this page use `careorg-a-acc`.

Using the API

```
POST <internal Nuts interface>/internal/vdr/v2/subject
Content-Type: application/json

{"subject": "careorg-a-acc"}
```

The response contains the subject ID and its DID documents:

```
{
  "subject": "careorg-a-acc",
  "documents": [
    {
      "id": "did:web:nuts.example.com:iam:657f064a-ebef-4f0f-aa87-88ed32db3142",
      ...
    }
  ]
}
```

Using Nuts Admin

1. Log in to Nuts Admin.
2. Click "Identities".
3. Click the plus sign on the right.
4. Enter the subject ID, e.g. `careorg-a-acc`.
5. Click "Create Identity".

Check that it worked

Other parties must be able to resolve the DID from the internet. Test that with a public DID resolver:

1. Take the DID from the `id` field in the response above, e.g. `did:web:nuts.example.com:iam:657f064a-ebef-4f0f-aa87-88ed32db3142`.
2. Paste it into the Universal Resolver at https://dev.uniresolver.io/ and click "Resolve".
3. It should return the DID document. If it fails, check that the node's public interface is reachable from the internet through your reverse proxy (see chapter "Deploy"). The resolver fetches the document from `https://nuts.example.com/iam/657f064a-ebef-4f0f-aa87-88ed32db3142/did.json`, so you can also open that URL from outside your network.

You need this DID in the next step, to issue the X509Credential. To look up a subject's DIDs later, use `GET <internal Nuts interface>/internal/vdr/v2/subject/careorg-a-acc`.

API reference: Verifiable Data Registry (v2)

Get an X509Credential from a UZI certificate

Many use cases identify care organizations by their URA number. The CIBG doesn't issue Verifiable Credentials with the URA yet, so you create one yourself from the care organization's UZI server certificate: an X509Credential.

The node uses the X509Credential to:

- register the care organization on the Discovery Service, so other parties can find it;
- prove the care organization's identity when it requests an access token from another party.

The X509Credential is:

- **issued by** the UZI certificate: the issuer is a did:x509 DID derived from the certificate, and the credential is signed with the certificate's private key;
- **issued to** the care organization's did:web DID, which belongs to its subject on your Nuts node. That subject holds the credential in its wallet.

It contains the UZI certificate, and the URA, name and city taken from it. It does **not** contain the private key.

For background, such as how the URA ends up in the credential and which certificate authority a use case trusts, see X509Credential with UZI server certificates. The credential's definition: X509Credential in the credential catalog.

Do this once per care organization, per environment.

Who does what

The UZI certificate and its private key belong to the care organization. Ideally, the private key never leaves the care organization:

Step	Who
Create the subject and give its did:web DID to the care organization	Vendor
Create the X509Credential with the UZI certificate and private key	Care organization

Step	Who
Send the X509Credential (a JWT, without the private key) to the vendor	Care organization
Load the X509Credential into the wallet	Vendor

In practice, the vendor often already manages the care organization's UZI certificate, e.g. for client TLS connections. The vendor then has the private key anyway, and can create the X509Credential itself. The steps below work for the care organization and the vendor alike.

Before you start

- The care organization has a subject, and you have its did:web DID from the vendor (see "Create a subject").
- The care organization's UZI server certificate from the CIBG:
  - the full certificate chain (root CA up to the certificate itself) as a PEM file;
  - the unencrypted private key of the certificate as a PEM file.
- A tool to create the credential. Choose one:
  - The `nutsfoundation/go-didx509-toolkit` Docker image (source), to create the credential by hand. The steps below use this tool.
  - The Java library uzi-did-x509-issuer-java, to create the credential from your EHR or another Java application, without Docker. See its README for usage.

Create the credential

Put the certificate chain (`certificate.pem`) and private key (`privatekey.pem`) in a directory called `certs`. From the directory that contains `certs`, run:

```
docker run --rm -v "$(pwd)/certs:/certs" \
  nutsfoundation/go-didx509-toolkit:1.2.0 \
  vc /certs/certificate.pem /certs/privatekey.pem \
  "<CA name>" \
  <did>
```

- `<did>` is the care organization's did:web DID, from "Create a subject".
- `<CA name>` is the full name (subject DN) of the CIBG certificate authority that issued the UZI server certificate. It depends on the type of certificate:

Certificate	CA (`CN` part of the name)
Production, G1	`UZI-register Private Server CA G1`
Production, G4 (from 12 November 2026)	`UZI Server - G4 PKIo Priv G-TLS SYS - 2025`
CIBG test, G1	`TEST UZI-register Private Server CA G1`

Certificate	CA (`CN` part of the name)
CIBG acceptance, G4	`ACCEPTATIE UZI Server - G4 Priv G-TLS SYS - 2025`

The easiest way to get the exact name: run the command with `"<CA name>"` as it is. The tool then fails with `ca-fingerprint certificate not found in chain` and lists the names of all certificates in your chain under `Candidates:`. Copy the one for the CA and run the command again.

Check https://hub.docker.com/r/nutsfoundation/go-didx509-toolkit/tags for the latest version.

The output is the X509Credential as a JWT (a long string starting with `eyJ`). Send it to the vendor, who loads it into the care organization's wallet (see "Load the credential into the wallet").

Non-production environments

For non-production environments, the CIBG issues test certificates whose name, URA and city don't match the real care organization. Where the use case allows it, you can instead create a fake UZI server certificate with a test CA:

1. Check out https://github.com/nuts-foundation/go-didx509-toolkit. You need Bash and `openssl`.
2. In the `test_ca` directory, run:

   ```
   ./issue-cert.sh <domain (CN)> <name (O)> <city (L)> <uzi> <ura> <agb>
   ```

   The domain, UZI and AGB values can be anything; use cases don't use them.
3. The certificate chain and private key are written to the `out` directory. Copy them to your `certs` directory.
4. Create the credential as above, but with `"CN=Fake UZI Root CA"` as the CA:

   ```
   docker run --rm -v "$(pwd)/certs:/certs" \
     nutsfoundation/go-didx509-toolkit:1.2.0 \
     vc /certs/<domain>-chain.pem /certs/<domain>.key \
     "CN=Fake UZI Root CA" \
     <did>
   ```

Which CAs an environment accepts is set in the use case's Discovery Service definition.

> **360-graden-clientbeeld**
>
> The X509Credential is the only credential a care organization needs to obtain. Accepted certificates per environment:

Environment	Accepted certificates
Development, test	Fake UZI certificates and CIBG test certificates (G1 and acceptance G4)
Acceptance	CIBG test certificates (G1 and acceptance G4) and real G1 certificates. Not fake ones, and not real G4 ones.
Production	Real CIBG certificates only (G1 and G4)

Lifecycle

The X509Credential is only valid as long as the UZI certificate it was created from. Create and load a new X509Credential:

- before the UZI certificate expires, with the renewed certificate;
- when the UZI certificate is replaced or revoked;
- when the care organization's details in the certificate change (name, city).

Load the new credential as described in "Load the credential into the wallet", then remove the old one (see "Renew credentials").

Load the credential into the wallet

Load the X509Credential into the care organization's wallet on the Nuts node. The node uses credentials from the wallet when it registers on the Discovery Service and when it requests access tokens.

The same steps work for any other credential you receive for the care organization.

Before you start

- The care organization has a subject (see "Create a subject").
- You have the X509Credential as a JWT (see "Get an X509Credential from a UZI certificate").

Using the API

```
POST <internal Nuts interface>/internal/vcr/v2/holder/careorg-a-acc/vc
Content-Type: application/json


"eyJ..."
```

- Replace `careorg-a-acc` with the subject ID.
- The body is the JWT as a JSON string: the JWT in double quotes.
- The node returns `204 No Content` on success. It returns `400 Bad Request` if the credential is invalid, for example when it was issued to a DID that doesn't belong to this subject, or when it has expired.

Using Nuts Admin

1. Log in to Nuts Admin.
2. Click "Identities".
3. Click the care organization.
4. Under "Credentials in Wallet", click "Upload".
5. Paste the credential (the JWT) in the input field.
6. Click "Upload Credential".

Check that it worked

List the credentials in the wallet:

```
GET <internal Nuts interface>/internal/vcr/v2/holder/careorg-a-acc/vc
```

The result should contain a credential with type `X509Credential`.

API reference: Verifiable Credential Registry (v2)

Obtaining credentials

Some use cases require credentials besides the X509Credential. Check the use case's Discovery Service and policy definitions to see which credentials you need. Credential types are defined in the credential catalog.

> **360-graden-clientbeeld**
>
> No other credentials are needed. You can skip this page.

A credential is issued in one of two ways, depending on the use case:

Issued by	When	What you do
A trusted third party	The use case names an issuer, e.g. VZVZ or Vektis	Receive the credential from the issuer and load it into the wallet
The care organization itself (self-issued)	The use case allows it, e.g. because there's no trusted issuer yet	Issue it with your Nuts node, then load it into the wallet

Self-issue a credential

The steps below use the HealthcareProviderRoleTypeCredential as example, from the Huisartsinzage use case.

The HealthcareProviderRoleTypeCredential states the type of care organization, e.g. `H1` (huisartsinstelling) or `X3` (verplegings- of verzorgingsinstelling). Huisartsinzage requires it. VZVZ might issue it in the future; until then, care organizations issue it themselves.

Before you start

- The care organization has a subject and a did:web DID (see "Create a subject").

Issue the credential

```
POST <internal Nuts interface>/internal/vcr/v2/issuer/vc
Content-Type: application/json

{
  "@context": "https://www.w3.org/2018/credentials/v1",
```

```
  "type": "HealthcareProviderRoleTypeCredential",
  "issuer": "did:web:nuts.example.com:iam:657f064a-ebef-4f0f-aa87-88ed32db3142",
  "credentialSubject": {
    "id": "did:web:nuts.example.com:iam:657f064a-ebef-4f0f-aa87-88ed32db3142",
    "roleCodeNL": "X3"
  },
  "expirationDate": "2027-10-27T00:00:00Z",
  "format": "jwt_vc"
}
```

- `issuer` and `credentialSubject.id` are both the care organization's DID: it issues the credential to itself.
- `format` must be `jwt_vc`. The default format (`ldp_vc`, JSON-LD) fails, because no JSON-LD context defines `roleCodeNL`.
- Set `expirationDate`. Without it, the node requires `withStatusList2021Revocation`.
- To make the credential revocable, add `"withStatusList2021Revocation": true`.

The response is the credential as a JWT, in double quotes.

Load the credential into the wallet

Issuing doesn't put the credential in the wallet. Load it as described in "Load the credential into the wallet", using the response from the step above as the request body.

Lifecycle

Issue and load a new credential before `expirationDate`, and when the care organization's type changes.

API reference: Verifiable Credential Registry (v2)

Join the use case

Do this once per care organization, per use case: register on the use case's Discovery Service so other parties can find you.

Register on the Discovery Service

Register the care organization on the use case's Discovery Service, so other parties can find it and its API endpoints. In the Nuts node this is called "activating" the Discovery Service for a subject.

Do this once per care organization, per use case. The node then keeps the registration up to date by itself.

Before you start

- The node has the use case's Discovery Service definition (see chapter "Configure the node for a use case").
- The care organization has a subject (see "Create a subject").
- The wallet has the credentials the Discovery Service definition requires (see "Load the credential into the wallet").
- You know which registration parameters the use case requires, see below.

Registration parameters

Registration parameters are values you publish with the registration, typically API endpoints. The use case says which ones you need. The node puts them in a DiscoveryRegistrationCredential, so you don't need to issue that credential yourself.

The node always adds `authServerURL`: your Nuts node's OAuth2 authorization server URL for this subject.

360-graden-clientbeeld

The wallet only needs the X509Credential. Registration parameters:

Parameter	Required	Value
`fhirR4BaseURL`	At least one of the two	Base URL of your FHIR R4 API
`fhirStu3BaseURL`	At least one of the two	Base URL of your FHIR STU3 API

The service ID is `360-graden`.

Using the API

```
POST <internal Nuts interface>/internal/discovery/v1/<service ID>/<subject ID>
Content-Type: application/json

{
  "registrationParameters": {
    "<parameter>": "<value>"
  }
}
```

- `<service ID>` is the `id` field of the Discovery Service definition.
- `<subject ID>` is the care organization's subject ID, e.g. `careorg-a-acc`.

360-graden-clientbeeld

```
POST <internal Nuts interface>/internal/discovery/v1/360-graden/careorg-a-acc
Content-Type: application/json

{
  "registrationParameters": {
    "fhirR4BaseURL": "https://fhir.example.com/r4"
  }
}
```

The node registers right away:

Response	Meaning	What to do
`200 OK`	Registered. The node refreshes the registration by itself before it expires.	Nothing
`412 Precondition Failed`	Not registered. The response says why, e.g. missing credentials or the Discovery Server can't be reached. The node does **not** retry by itself.	Fix the cause and call the API again
`404 Not Found`	Unknown service ID or subject ID	Check both IDs

Check that it worked

Get the registration status:

```
GET <internal Nuts interface>/internal/discovery/v1/<service ID>/<subject ID>
```

- `activated` should be `true` and `status` should be `active`.
- If `status` is `error`, `error` says what went wrong, e.g. a refresh that failed.

Then check that others can find you, by searching for your own care organization, e.g. by name:

```
GET <internal Nuts interface>/internal/discovery/v1/<service ID>?<path in the
credential>=<value>
```

> **360-graden-clientbeeld**
>
> The name is in the X509Credential:
>
> ```
> GET <internal Nuts interface>/internal/discovery/v1/360-
> graden?credentialSubject.subject.O=Zorgorganisatie%20A*
> ```

- The query parameter is a path in the credential (without `$.`), not a field ID from the definition.
- Write the value with the same upper and lower case as in the credential. `*` is a wildcard at the start or end.
- Your care organization should be in the result, with your DID in `credential_subject_id` and your registration parameters in `registrationParameters`.

See "Find care organizations and their endpoints" for more search options.

Changing or removing the registration

- **Change the registration parameters**, e.g. a new FHIR URL: call the `POST` again with the new values.
- **Remove the registration**:

  ```
  DELETE <internal Nuts interface>/internal/discovery/v1/<service ID>/<subject ID>
  ```

  `200 OK` means it was removed. `202 Accepted` means the node stopped refreshing, but couldn't remove the registration from the Discovery Server; it disappears there when it expires.

API reference: Discovery Service

Outbound: querying another party

Find another care organization, request an access token, and query its FHIR API.

Find care organizations and their endpoints

Before you can query another care organization, you need its API endpoint and its authorization server URL. Find both by searching the use case's Discovery Service through your own Nuts node.

Before you start

- The node has the use case's Discovery Service definition (see chapter "Configure the node for a use case").

Search

```
GET <internal Nuts interface>/internal/discovery/v1/<service ID>?<path in the credential>=<value>
```

- `<service ID>` is the `id` field of the Discovery Service definition.
- Without query parameters, you get all registered care organizations.

Query parameters are paths in the registered credentials:

- Write the path without `$.`, e.g. `credentialSubject.subject.O`.
- Field IDs from the definition, like `organization_name`, don't work as query parameters.
- Write the value with the same upper and lower case as in the credential. Use `*` at the start or end of a value as a wildcard.
- With several query parameters, all of them must match the same credential.
- Only text fields can be searched; arrays, numbers and true/false values can't.

Which paths you can search on depends on the credentials the use case requires.

360-graden-clientbeeld

Name and city are in the X509Credential, so you can combine them:

Search by	Query parameter
Name	`credentialSubject.subject.O=Zorgorganisatie*`
City	`credentialSubject.subject.L=Utrecht`

```
GET <internal Nuts interface>/internal/discovery/v1/360-graden?credentialSubject.subject.O=Zorgorganisatie*
```

Your node searches its own copy of the Discovery Service. It updates that copy every 10 minutes by default (`discovery.client.refreshinterval`), so a new registration can take that long to show up.

The result

Part	What it is
`credential_subject_id`	The care organization's DID
`fields`	Values taken from the credentials, named by the field IDs in the Discovery Service definition
`registrationParameters`	The values the care organization registered, e.g. its API endpoints
`registrationParameters.authServerURL`	The authorization server URL. You need it to request an access token (see "Request an access token")

Which fields and registration parameters you get depends on the use case.

360-graden-clientbeeld


```
[
  {
    "id": "did:web:nuts.other.example:iam:2f1b7c3e-0d1a-4a7e-9a55-4b8f3c2e9d10#7c4e...",
    "credential_subject_id": "did:web:nuts.other.example:iam:2f1b7c3e-0d1a-4a7e-9a55-4b8f3c2e9d10",
    "fields": {
      "organization_id": "did:web:nuts.other.example:iam:2f1b7c3e-0d1a-4a7e-9a55-4b8f3c2e9d10",
      "organization_name": "Zorgorganisatie B",
      "organization_city": "Utrecht",
      "organization_ura": "87654321",
      "authorization_server_url": "https://nuts.other.example/oauth2/careorg-b-acc",
      "fhir_base_url_r4": "https://fhir.other.example/r4"
    },
    "registrationParameters": {
      "authServerURL": "https://nuts.other.example/oauth2/careorg-b-acc",
      "fhirR4BaseURL": "https://fhir.other.example/r4"
```

```
    },
    "vp": { ... }
  }
]
```

`organization_ura` is the bare URA, because the definition's pattern has a capture group.

API reference: Discovery Service

Request an access token

To query another care organization's API, you need an access token from that organization's authorization server. Your Nuts node requests it for you: it shows the authorization server the credentials the use case requires.

Before you start

- You found the other care organization and its `authServerURL` (see "Find care organizations and their endpoints").
- Your care organization's wallet has the credentials the use case's policy requires (see "Load the credential into the wallet").
- You know the scope. The use case sets it.

> **360-graden-clientbeeld**
>
> - The wallet needs the X509Credential.
> - The scope is `360-graden`.
> - Each request needs a NutsEmployeeCredential for the logged-in user (see below).

Request the token

```
POST <internal Nuts interface>/internal/auth/v2/<subject ID>/request-service-access-token
Content-Type: application/json

{
  "authorization_server": "<authServerURL of the other care organization>",
  "scope": "<scope>",
  "credentials": [ <credentials that aren't in the wallet, if the use case requires them> ]
}
```

Field	Value
`<subject ID>` (in the URL)	The subject ID of your care organization: the one making the request, e.g. `careorg-a-acc`
`authorization_server`	The other care organization's `authServerURL`, from the Discovery Service search result

Field	Value
`scope`	Set by the use case
`credentials`	Optional. Credentials to present that aren't in the wallet, typically about the user. See below
`token_type`	Optional. `DPoP` (default) or `Bearer`. Use DPoP: a stolen DPoP token can't be used without your node's key

The user's identity

Some use cases require a credential about the logged-in user, e.g. a NutsEmployeeCredential. Pass it in `credentials`. You don't sign it, and you leave out `issuer` and `credentialSubject.id`: the node fills them in with your care organization's DID. The node's signature on the whole request protects it.

360-graden-clientbeeld

A NutsEmployeeCredential is required, for the other party's audit log:

Field	Value
`name`	The user's name
`roleName`	The user's role, e.g. "Verpleegkundige"
`identifier`	The user's ID in your system

```
POST <internal Nuts interface>/internal/auth/v2/careorg-a-acc/request-service-access-token
Content-Type: application/json

{
  "authorization_server": "https://nuts.other.example/oauth2/careorg-b-acc",
  "scope": "360-graden",
  "credentials": [
    {
      "@context": [
        "https://www.w3.org/2018/credentials/v1",
        "https://nuts.nl/credentials/v1"
      ],
      "type": ["VerifiableCredential", "NutsEmployeeCredential"],
      "credentialSubject": {
        "name": "J. de Vries",
```

```
        "roleName": "Verpleegkundige",
        "identifier": "123456"
      }
    }
  ]
}
```

The response

```
{
  "access_token": "eyJhbGciOiJSUzI1NiIsInR5cCI6Ikp...",
  "token_type": "DPoP",
  "dpop_kid": "did:web:nuts.example.com:iam:657f064a-ebef-4f0f-aa87-88ed32db3142#<key id>",
  "scope": "<scope>",
  "expires_in": 900
}
```

- Use `access_token` in the `Authorization` header when you call the other party's FHIR API (or other data API).
- For a DPoP token, you also need a DPoP header for every request, signed with the key in `dpop_kid` (see "Make the FHIR call").
- The node caches tokens: calling this API again returns the same token while it's valid, so you don't have to cache it yourself.

Errors

Response	Meaning
`400 Bad Request`	A parameter is wrong, or the authorization server returned an OAuth error
`412 Precondition Failed`	Your wallet doesn't have the credentials the policy requires
`502 Bad Gateway`	The other party's authorization server returned an error
`503 Service Unavailable`	The other party's authorization server can't be reached

API reference: Auth (v2)

Make the FHIR call

With an access token, you can query the other care organization's FHIR API (or other data API).

The token is either a DPoP token (the default) or a Bearer token, depending on `token_type` in your token request. DPoP is optional, but more secure: every request also needs a DPoP header, which proves the request comes from you, the party the token was issued to. A stolen DPoP token is useless without your node's key. Your Nuts node creates the DPoP header for you.

Before you start

- You have an access token and its `dpop_kid` (see "Request an access token").
- You have the other care organization's API endpoint, from its `registrationParameters` (see "Find care organizations and their endpoints").
- Your client can connect to the other party's API. Data endpoints are protected with mTLS (see "Endpoints and TLS certificates").

1. Create the DPoP header (DPoP tokens only)

Create a new DPoP header for **every** request. It's only valid for one HTTP method and URL, and can only be used once.

```
POST <internal Nuts interface>/internal/auth/v2/dpop/<dpop_kid, URL-encoded>
Content-Type: application/json

{
  "htm": "<HTTP method>",
  "htu": "<URL of the request>",
  "token": "<access token>"
}
```

Field	Value
`<dpop_kid>` (in the URL)	`dpop_kid` from the token response, URL-encoded once, e.g. `#` becomes `%23`
`htm`	The HTTP method of the request you're about to make, e.g. `GET`
`htu`	The URL of the request you're about to make. Query parameters are ignored.

Field	Value
`token`	The access token

The response:

```
{
  "dpop": "eyJhbGciOiJFUzI1NiIsInR5cCI6ImRwb3Arand0In0..."
}
```

2. Make the request

Send the access token in the `Authorization` header with the type `DPoP`, and the DPoP header in `DPoP`:

```
<HTTP method> <URL of the request>
Authorization: DPoP <access token>
DPoP: <dpop from step 1>
```

For a Bearer token, skip step 1 and send `Authorization: Bearer <access token>`.

Errors

Response from the other party	What to do
`401 Unauthorized`	The token expired, or the DPoP header doesn't match the request. Check `htm` and `htu`. If the token expired, request a new one. If it hasn't expired, the other party may have lost it: request a new token with the header `Cache-Control: no-cache`, which skips your node's token cache.
`403 Forbidden`	The other party doesn't allow this query, e.g. no legal basis such as consent. Check the use case's rules.

360-graden-clientbeeld

Use `fhirR4BaseURL` or `fhirStu3BaseURL` from the search result, depending on which FHIR version the other party offers. The examples below use R4, with base URL `https://fhir.other.example/r4`.

1. Get the CapabilityStatement. It says which queries the other party supports. It doesn't need an access token, and you need it before any query other than Patient.

```
GET https://fhir.other.example/r4/metadata
```

2. Find the patient by BSN. Only POST-based search is allowed. Create the DPoP header with `htm` `POST` and `htu` `https://fhir.other.example/r4/Patient/_search`.

```
POST https://fhir.other.example/r4/Patient/_search
Content-Type: application/x-www-form-urlencoded
Authorization: DPoP <access token>
DPoP: <dpop>

identifier=http://fhir.nl/fhir/NamingSystem/bsn|123456789
```

The result has the patient's ID at the other party, e.g. `Patient/abc123`.

3. Query the data. For example, the patient's allergies. Create a new DPoP header with `htm` `GET` and `htu` `https://fhir.other.example/r4/AllergyIntolerance`.

```
GET
https://fhir.other.example/r4/AllergyIntolerance?patient=abc123&_profile=http://nictiz.nl/fhir/StructureDefinition/nl-core-AllergyIntolerance,http://nictiz.nl/fhir/StructureDefinition/zib-AllergyIntolerance
Authorization: DPoP <access token>
DPoP: <dpop>
```

- Queries select resources by FHIR profile (`_profile`). Volume 3 of the IG lists the query for each zib and FHIR version.
- Results are paged: use `_count` (maximum 100) and follow the bundle's `next` link.
- The order of requests is in volume 2b of the IG.

API reference: Auth (v2)

Inbound: handling queries

Check the access token on incoming requests and decide whether to allow the query.

Validate the access token

When another care organization queries your FHIR API (or other data API), the request carries an access token your Nuts node issued. Your API (or the API gateway in front of it) must check that token on every request, then decide whether to allow the query.

The Nuts node checks the token and tells you who is asking. Whether that party may see the data is decided by your application.

Before you start

- The node has the use case's policy definition (see chapter "Configure the node for a use case"). The policy says which credentials another party must present to get a token for a scope. Without it, the node doesn't issue tokens for that scope.
- Your care organization is registered on the Discovery Service, so others can find your authorization server (see "Register on the Discovery Service").

What an incoming request looks like

For a DPoP token, the request has two headers:

```
<HTTP method> <URL of the request>
Authorization: DPoP eyJhbGciOiJSUzI1NiIsInR5cCI6Ikp...
DPoP: eyJhbGciOiJFUzI1NiIsInR5cCI6ImRwb3Arand0In0...
```

For a Bearer token, there's only `Authorization: Bearer <token>`.

1. Introspect the token

```
POST <internal Nuts interface>/internal/auth/v2/accesstoken/introspect
Content-Type: application/x-www-form-urlencoded

token=eyJhbGciOiJSUzI1NiIsInR5cCI6Ikp...
```

The result has the standard OAuth2 fields, plus the fields from the use case's policy definition:

Field	What it is
`active`	Whether the token is valid

Field	What it is
`scope`	The scope the token was issued for
`iss`	The authorization server that issued the token: your node, for one of your care organizations
`client_id`	The client ID the requester gave in its token request. It isn't verified, so don't use it to identify the requester: use the fields from the credentials instead, e.g. `organization_ura`.
`exp`	When the token expires (seconds since 1970)
`cnf.jkt`	DPoP tokens only: the thumbprint of the requester's DPoP key
Other fields	Values from the credentials the requester presented, named by the field IDs in the policy definition

360-graden-clientbeeld

```
{
  "active": true,
  "iss": "https://nuts.example.com/oauth2/careorg-a-acc",
  "client_id": "https://nuts.other.example/oauth2/careorg-b-acc",
  "scope": "360-graden",
  "exp": 1793110000,
  "cnf": { "jkt": "jlkhnp87453slfhansdhf" },
  "organization_ura": "87654321",
  "organization_name": "Zorgorganisatie B",
  "organization_city": "Utrecht",
  "employee_name": "J. de Vries",
  "employee_role": "Verpleegkundige",
  "employee_identifier": "123456"
}
```

- If `active` is `false`, reject the request with `401 Unauthorized`. The token is expired, invalid or unknown.
- Check that `scope` is the scope of the use case the request belongs to.
- Check that `iss` is the authorization server of the care organization whose data is requested, e.g. `https://nuts.example.com/oauth2/careorg-a-acc`. Your node hosts several care organizations and introspects tokens for all of them, so without this check a token issued for one care organization could be used to query another.
- `cnf.jkt` is only there for DPoP tokens. You need it in the next step.

Need the presented credentials themselves, e.g. for your audit log? Use `POST <internal Nuts interface>/internal/auth/v2/accesstoken/introspect_extended`: same result, plus the Verifiable Presentations.

2. Check the DPoP header (DPoP tokens only)

A DPoP token is only valid together with a DPoP header that the requester signed for this request:

```
POST <internal Nuts interface>/internal/auth/v2/dpop/validate
Content-Type: application/json

{
  "dpop_proof": "eyJhbGciOiJFUzI1NiIsInR5cCI6ImRwb3Arand0In0...",
  "method": "<HTTP method of the request>",
  "url": "<URL of the request>",
  "thumbprint": "jlkhnp87453slfhansdhf",
  "token": "eyJhbGciOiJSUzI1NiIsInR5cCI6Ikp..."
}
```

Field	Value
`dpop_proof`	The `DPoP` header of the incoming request
`method`	The HTTP method of the incoming request
`url`	The URL of the incoming request, as the requester called it. Query parameters are ignored.
`thumbprint`	`cnf.jkt` from the introspection result
`token`	The access token from the `Authorization` header

The node answers with `200 OK` and whether the DPoP header is valid:

```
{
  "valid": false,
  "reason": "DPoP proof is expired"
}
```

- `valid: true`: the DPoP header belongs to this token and this request.
- `valid: false`: reject the request with `401 Unauthorized`. `reason` says why.

Use the URL as the requester called it, not as your API sees it after a reverse proxy rewrote it. Otherwise the check fails.

3. Decide whether to allow the query

The Nuts node doesn't decide whether the requester may see this data. Your application does, using the introspection result and the rules of the use case.

The use case's IG describes the rules, e.g. which legal basis is needed (such as consent registered in Mitz), and what the request must contain.

You don't have to build this from scratch: **Nuts Knooppunt** has a policy decision point (PDP). Your reverse proxy or API gateway sends it the introspection result and the request details, and gets back allow or deny. It checks things like the FHIR CapabilityStatement, required search parameters and Mitz consent. It has built-in policies per scope, e.g. for BgZ, eOverdracht and PZP. See using the PDP in the Knooppunt docs.

> **360-graden-clientbeeld**
>
> The rules are in volume 2a of the IG. In short:
>
> - The request must fit the use case's FHIR CapabilityStatement.
> - The requesting organization (`organization_ura`) must be identified, and the user (`employee_identifier`, `employee_role`) must be present for the audit log.
> - There must be a legal basis: consent registered in Mitz, the requester is in the patient's CareTeam, or another legal basis.
> - The request must be about one patient: a BSN in `/Patient/_search`, otherwise a reference to the Patient.
>
> The IG has an OPA policy that implements these rules. It isn't used in production yet, but can be a starting point. Knooppunt has no built-in 360-graden policy.

Log every request with the requester's organization and user, as the use case requires.

API reference: Auth (v2)

Test your setup

Check each part of your setup, from the node itself to a query with another party.

Test your setup step by step

Check your setup in the order you built it. Each check tells you which part works; if one fails, fix it before going on.

#	Check	Tests
1	The node runs	Deployment
2	Others can reach your node	Reverse proxy, TLS, URL rewrite
3	The wallet has the credentials	Credentials
4	You're registered and can be found	Discovery Service registration
5	You can get an access token from yourself	Policy definition, credentials, token flow, DPoP
6	You can query another party	Everything, with a real other party

1. The node runs

```
GET <internal Nuts interface>/health
```

Should return `200 OK` with status `UP`. If not, check the node's log.

2. Others can reach your node

From **outside** your network:

- Resolve a subject's DID with the Universal Resolver (see "Create a subject").
- Open your OAuth2 metadata: `https://<your node>/.well-known/oauth-authorization-server/oauth2/<subject ID>`. It should return JSON with `"issuer": "https://<your node>/oauth2/<subject ID>"`. If your node runs on a subpath, use the URL from "Check that it works" in "Configure url rewrite for OAuth" instead.

If this fails, check your reverse proxy, TLS certificate and URL rewrite (see chapter "Deploy").

3. The wallet has the credentials

```
GET <internal Nuts interface>/internal/vcr/v2/holder/<subject ID>/vc
```

The result should contain every credential the use case requires (see "Load the credential into the wallet").

4. You're registered and can be found

```
GET <internal Nuts interface>/internal/discovery/v1/<service ID>/<subject ID>
```

`activated` should be `true` and `status` `active`. Then search for your own care organization (see "Register on the Discovery Service").

5. Get an access token from yourself

Request a token from your own authorization server, as if you were another party. This tests your policy definition, your credentials and the whole token flow.

```
POST <internal Nuts interface>/internal/auth/v2/<subject ID>/request-service-access-token
Content-Type: application/json

{
  "authorization_server": "https://<your node>/oauth2/<subject ID>",
  "scope": "<scope>",
  "credentials": [ <user credentials, if the use case requires them> ]
}
```

`authorization_server` is your own `authServerURL`. Your node reaches it through its public URL, so this also tests your reverse proxy.

The result is an access token (see "Request an access token"). Then check it as you would an incoming request (see "Validate the access token"):

1. Introspect the token. `active` should be `true`, and the fields from the policy definition should be there.
2. Create a DPoP header for a test URL (see "Make the FHIR call"), and validate it with the same method and URL. `valid` should be `true`.
3. Validate the same DPoP header again. Now `valid` should be `false`, with reason `jti already used`: a DPoP header can only be used once.

Response to the token request	Likely cause
`412 Precondition Failed`	The wallet doesn't have the credentials the policy requires
`400 Bad Request`	Wrong scope (no policy for it), or a wrong `credentials` value
`503 Service Unavailable`	Your node can't reach its own public URL

360-graden-clientbeeld

Use scope `360-graden`, and pass a NutsEmployeeCredential with test values:

```
{
  "authorization_server": "https://<your node>/oauth2/careorg-a-acc",
  "scope": "360-graden",
  "credentials": [
    {
      "@context": ["https://www.w3.org/2018/credentials/v1",
"https://nuts.nl/credentials/v1"],
      "type": ["VerifiableCredential", "NutsEmployeeCredential"],
      "credentialSubject": { "name": "Test User", "roleName": "Tester", "identifier":
"test-1" }
    }
  ]
}
```

The introspection result should contain `organization_id`, `organization_ura`, `organization_name`, `organization_city`, `employee_name`, `employee_role` and `employee_identifier`.

6. Query another party

In a test environment, agree with another vendor to test with each other's care organizations:

1. Find their care organization (see "Find care organizations and their endpoints").
2. Request a token from their authorization server, and make a query (see chapter "Outbound: querying another party").
3. Let them do the same with yours, and check your logs.

Running it

Keep credentials valid, monitor and back up the node, and fix common problems.

Renew credentials

Credentials expire. When one expires, the node no longer uses it, and the care organization can't register on the Discovery Service or get access tokens anymore. Renew them in time.

What to renew, and when

What	When	How
X509Credential	Before the UZI server certificate expires, or when it's replaced or revoked	Create a new one from the renewed certificate (see "Get an X509Credential from a UZI certificate")
Self-issued credentials	Before their `expirationDate`	Issue a new one (see "Obtaining credentials")
Credentials from a trusted third party	Before they expire	Ask the issuer for a new one
Discovery Service registration	Never: the node refreshes it by itself	—

Keep a list of expiry dates per care organization, or check them regularly (see "Find expiring credentials" below).

Replace a credential

1. Load the new credential into the wallet (see "Load the credential into the wallet").
2. Remove the old credential from the wallet, so the node doesn't pick it by mistake:

   ```
   DELETE <internal Nuts interface>/internal/vcr/v2/holder/<subject ID>/vc/<credential ID, URL-encoded>
   ```

   Find the credential ID (`id`) by listing the wallet: `GET <internal Nuts interface>/internal/vcr/v2/holder/<subject ID>/vc`.

The node uses the new credential right away for new access tokens. It uses it for the Discovery Service registration at the next refresh. To update the registration right away, call the registration `POST` again (see "Register on the Discovery Service").

Find expiring credentials

List the wallet and look at each credential's `expirationDate` (or `exp` for credentials in JWT form):

```
GET <internal Nuts interface>/internal/vcr/v2/holder/<subject ID>/vc
```

This only returns valid credentials: expired ones are left out.

Operate the node

Monitoring

Endpoint (internal interface)	Use it for
`/health`	Health checks by Docker or Kubernetes. Returns `200 OK` when the node is healthy.
`/status/diagnostics`	Details about the node's state, for administrators
`/metrics`	Metrics in Prometheus format

See monitoring in the Nuts node documentation.

Backups

Back up both, and test that you can restore them:

- the SQL database (`storage.sql.connection`);
- the private keys (`crypto.storage`, e.g. HashiCorp Vault or Azure Key Vault).

Without the private keys, the node can't use its DIDs anymore, and every care organization needs a new subject, new credentials and a new registration.

Upgrades

- Read the release notes before you upgrade.
- Upgrade test and acceptance first, then production.
- Keep a fixed version in your deployment (see "Deploy the Nuts node").

Troubleshooting

Problem	Likely cause	What to do
The node doesn't start	A Discovery Service definition can't be read	The log says which file. Use the files from the use case's IG, unchanged.
A DID doesn't resolve from the internet	Reverse proxy, TLS or DNS	See "Endpoints and TLS certificates" and check 2 in "Test your setup step by step"

Problem	Likely cause	What to do
Discovery registration returns `412`	Missing or expired credentials, or the Discovery Server can't be reached	The response says which. Fix it and call the registration `POST` again: the node doesn't retry by itself.
Registration status is `error`	A refresh failed	`error` in the status says why. Fix it; the node retries at the next refresh.
You can't find a care organization that just registered	Your node updates its copy of the Discovery Service every 10 minutes by default	Wait, or check `discovery.client.refreshinterval`
Token request returns `412`	Your wallet doesn't have the credentials the other party's policy requires	Check the wallet and the use case's policy definition
Token request returns `503`	The other party's authorization server can't be reached	Check the `authServerURL` and your outbound network
The other party returns `401` for a token that hasn't expired	Their authorization server may have lost the token, e.g. after a restart	Request a new token with the header `Cache-Control: no-cache`, which skips your node's token cache. Don't do this by default: it adds load on both sides.
DPoP validation fails for incoming requests	The URL your API sees differs from the URL the requester used, e.g. after a reverse proxy rewrote it	Validate with the URL as the requester called it (see "Validate the access token")

Still stuck? Set `verbosity` to `debug` for more detail in the log, and ask on the Nuts Slack.