Implementing a Nuts spec v1 use case

How to implement a Nuts spec v1 use case, such as eOverdracht and KIK-v: `did:nuts` DIDs, the Nuts network, and credentials such as NutsOrganizationCredential. Spec v1 needs the Nuts network, which we want to move away from. New use cases use spec v2: see Implementing a Nuts v2 Use Case.

- How to use this book
- Join the Nuts network
- Set up DIDs and services
- Register, find and authorize care organizations
- Authenticate users
- Request an access token
- Check an access token

How to use this book

Spec v1 use cases (e.g. eOverdracht) use `did:nuts` and the Nuts network (gRPC).

The pages describe Nuts node v6. Where v5 differs, a note starts with **v5:**. To upgrade, see Migrating from v5 to v6.

`<internal Nuts interface>` is the address of the node's internal interface, `127.0.0.1:8081` by default. **v5:** the interface you configured, `:1323` by default.

The pages show how eOverdracht does each step, in blue boxes marked **eOverdracht**. See the eOverdracht leveranciersspecificatie for the full specification. For other use cases, see their specification. Older documentation calls a use case a *Bolt*: a specification that turns a care process into technical requirements.

1. Set up your Nuts node

Follow the Nuts node documentation (on Read the Docs):

1. Running on Docker
2. Configuration. Set `didmethods: ["nuts"]`, and manage DIDs with the APIs in this book only, not with the VDR v2 API.
3. Join the Nuts network (in this book): TLS certificate, bootstrap nodes, node DID
4. Recommended deployment

2. Implement the use case

1. Set up DIDs and services: your vendor DID, a DID per care organization, endpoints
2. Register, find and authorize care organizations: NutsOrganizationCredential, NutsAuthorizationCredential
3. Authenticate users: login contracts, if the use case needs a user identity
4. Request an access token: call another care organization's endpoint
5. Check an access token: handle calls from other care organizations

API reference: Nuts node API. The OpenAPI files there can be used to generate client code.

Join the Nuts network

Spec v1 use cases exchange DID documents and credentials over the Nuts network (gRPC). This page connects your node to a network and gives it an identity: the *node DID*. The node DID is also your vendor DID in the rest of this book.

Prerequisites

- A running Nuts node, see Running on Docker.
- A network to join.
- A TLS certificate for your node, for use as client and as server, from a CA that the other nodes in the network trust.
- The address of one or more nodes in the network to connect to first: the *bootstrap nodes*.

Networks

The nodes in a network trust the same CAs, have a certificate from one of them, share the same first transaction and use the same versions of the network protocol. There are four networks:

Network	For	Certificates
development	Testing new features. Nodes run the newest, not yet released version	Open CA on GitHub, see step 1
stable	Building your integration and testing with other vendors. Nodes run a released version	Open CA on GitHub, see step 1
test	Acceptance testing with other vendors and customers. Nodes run a released version	PKIoverheid
production	Production	PKIoverheid

1. Get a TLS certificate

Your node only accepts connections from nodes with a certificate from a CA in its *trust store*: a PEM file with the CA certificates the network agreed on. See RFC008 Certificate Structure.

For the development and stable networks, the CA certificate and key are public. Issue a certificate yourself with nuts-development-network-ca:

```
git clone https://github.com/nuts-foundation/nuts-development-network-ca
cd nuts-development-network-ca
./issue-cert.sh development nuts.yourdomain.example
```

This gives three files:

File	Contains
`nuts.yourdomain.example-development.key`	The private key of your node
`nuts.yourdomain.example-development.pem`	The certificate of your node
`truststore-development.pem`	The trust store of the network

For the test and production networks, get a certificate from PKIoverheid Domein Private Services. Put the root certificate ("Stamcertificaat") and all intermediate certificates under "Domein Private Services" in your trust store, and no other certificates.

2. Configure the network

A bootstrap node is a normal Nuts node that other nodes can connect to. Ask a participant of the network for the gRPC address of its node (port `5555` by default), or ask in `#development` on the Nuts Slack which bootstrap nodes there are.

```
didmethods: ["nuts"]
tls:
  truststorefile: /path/to/truststore-development.pem
  certfile: /path/to/nuts.yourdomain.example-development.pem
  certkeyfile: /path/to/nuts.yourdomain.example-development.key
network:
  bootstrapnodes:
    - nuts-development.other-vendor.example:5555
```

See Configuration for all settings, and Recommended deployment for the proxy in front of your node. **v5:** leave out `didmethods`.

Without the `network` settings the node still starts, but alone: a network of one node. That is fine for local development.

3. Wait until your node is in sync

(Re)start your node. It connects to the bootstrap node and gets a copy of the state of the network: the transactions with the DID documents and credentials of all parties. They also contain the

addresses of the other nodes; after a restart your node connects to those too.

```
GET <internal Nuts interface>/status/diagnostics
```

The diagnostics show the transactions coming in. Your node is in sync when no new ones arrive. How long that takes depends on the size of the network and on the CPU, memory and bandwidth of your node.

4. Create your node DID

Some data, such as private credentials, is only exchanged with a node that has proven who it is. Your node does that with a DID it manages: the node DID.

```
POST <internal Nuts interface>/internal/vdr/v1/did
```

Store the `id` of the returned DID document: this is your node DID. The node creates a key for it and keeps it in its key storage. The DID controls its own document, and can issue credentials.

Then add the `NutsComm` endpoint: the gRPC address on which other nodes reach your node.

```
POST <internal Nuts interface>/internal/didman/v1/did/<node DID>/endpoint
```

```
{
  "type": "NutsComm",
  "endpoint": "grpc://nuts.yourdomain.example:5555"
}
```

- The address has the form `grpc://<host>:<port>`.
- The host must be exactly one of the DNS names (SAN) in the TLS certificate of your node. Otherwise other nodes cannot verify your node DID.
- Other nodes ignore addresses with an IP address or a reserved name (RFC2606, e.g. `localhost` or `example.com`).

Set the node DID in the configuration, and restart your node:

```
network:
  nodedid: <node DID>
```

Nodes of the same organization can share one node DID, e.g. in a cluster.

To send private credentials directly from node to node, the DID document also needs a `node-http-services-baseurl` endpoint. The node adds it by itself when it can: see Configuration.

5. Check the connection

```
GET <internal Nuts interface>/status/diagnostics
```

It shows which nodes your node found and which it is connected to, and that your node DID is configured.

Next: Set up DIDs and services.

Set up DIDs and services

In spec v1, each party is a `did:nuts` DID, and its endpoints are services in its DID document:

DID	Who	Controlled by	Contains
Vendor DID	You, the vendor	Itself	Your contact information, your endpoints (incl. `NutsComm`)
Care organization DID	Each of your customers	Itself: your node has its key. **v5:** your vendor DID	References to your endpoints, per use case

Your vendor DID is the node DID you created when you joined the Nuts network.

Store the DIDs in your CRM (the software in which you manage your customers), with the customer they belong to. The node stores the DID documents and keys. If you run a node for your own care organization only, you are both the vendor and the care organization.

> **Use case design (spec v1)**
>
> The use case specifies the service types (e.g. `eOverdracht-sender`) and the endpoints each service must contain (e.g. `oauth`, `fhir`).

1. Add your contact information

Other node operators use it to contact you when your node or a DID document has a problem:

```
PUT <internal Nuts interface>/internal/didman/v1/did/<vendor DID>/contactinfo
```

```
{
  "name": "<vendor name>",
  "phone": "<phone number>",
  "email": "<email address>",
  "website": "<website URL>"
}
```

The node adds it to your DID document as a service of type `node-contact-info`.

2. Add your endpoints

Add each endpoint you host once, to your vendor DID. The care organization DIDs refer to them.

```
POST <internal Nuts interface>/internal/didman/v1/did/<vendor DID>/endpoint
```

```
{
  "type": "<endpoint type>",
  "endpoint": "<URL>"
}
```

You choose the `type` (e.g. `production-oauth`); it must be unique within the DID document. Whether `endpoint` is a base URL or a full URL depends on the use case. Services can share an endpoint: e.g. one `oauth` endpoint for all use cases.

To change or remove an endpoint:

```
PUT <internal Nuts interface>/internal/didman/v1/did/<vendor DID>/endpoint/<endpoint type>
DELETE <internal Nuts interface>/internal/didman/v1/did/<vendor DID>/endpoint/<endpoint type>
```

An endpoint that other services still refer to can't be removed.

You can also add the service of a use case to your vendor DID, as a compound service (the request of step 4, with your vendor DID). A care organization DID then refers to that whole service, instead of to each endpoint. This is what most vendors do: to change an endpoint or add one, you only change your vendor DID.

3. Create a care organization DID

Create a DID for each care organization:

```
POST <internal Nuts interface>/internal/vdr/v1/did
```

The request has no body. **v5:** send this body, so your vendor DID controls the DID. v6 ignores it:

```
{
  "selfControl": false,
  "controllers": ["<vendor DID>"],
  "assertionMethod": true,
  "capabilityInvocation": false
}
```

The response is the DID document:

```
{
  "@context": ["https://www.w3.org/ns/did/v1"],
  "id": "did:nuts:2mF6KT6eiSx5y2fwTP4Y42yMUh91zGVkbu4KMARvCJz9",
  "verificationMethod": [
    {
      "id":
"did:nuts:2mF6KT6eiSx5y2fwTP4Y42yMUh91zGVkbu4KMARvCJz9#_TKzHv2jFIyvdTGF1Dsgwngfdg3SH6TpDv0Ta1a
OEkw",
      "controller": "did:nuts:2mF6KT6eiSx5y2fwTP4Y42yMUh91zGVkbu4KMARvCJz9",
      "type": "JsonWebKey2020",
      "publicKeyJwk": { "kty": "EC", "crv": "P-256", "x": "...", "y": "..." }
    }
  ],
  "assertionMethod":
["did:nuts:2mF6KT6eiSx5y2fwTP4Y42yMUh91zGVkbu4KMARvCJz9#_TKzHv2jFIyvdTGF1Dsgwngfdg3SH6TpDv0Ta1
aOEkw"],
  "capabilityInvocation": ["..."],
  "keyAgreement": ["..."],
  "service": []
}
```

- Store the `id`: the DID of the care organization.
- All keys are listed in `verificationMethod`. The other lists say what a key is used for.
- `assertionMethod`: the node signs access token requests and credentials for the care organization with this key.
- `capabilityInvocation`: this key can change the DID document.

To remove a care organization, deactivate its DID. This can't be undone:

```
DELETE <internal Nuts interface>/internal/vdr/v1/did/<care organization DID>
```

Then refer to your `NutsComm` endpoint. Without it, the care organization doesn't receive private credentials, such as NutsAuthorizationCredentials:

```
POST <internal Nuts interface>/internal/didman/v1/did/<care organization DID>/endpoint
```

```
{
  "type": "NutsComm",
  "endpoint": "<vendor DID>/serviceEndpoint?type=NutsComm"
}
```

Next, register the care organization with a NutsOrganizationCredential, so others can find it: see Register, find and authorize care organizations.

4. Enable a use case for a care organization

Add the service of the use case to the care organization DID, as a *compound service*: a service whose endpoints refer to the endpoints of your vendor DID.

```
POST <internal Nuts interface>/internal/didman/v1/did/<care organization DID>/compoundservice
```

```
{
  "type": "<service type>",
  "serviceEndpoint": {
    "<endpoint name>": "<vendor DID>/serviceEndpoint?type=<endpoint type>"
  }
}
```

- `type` and the endpoint names come from the use case; add all endpoints it requires.
- `<endpoint type>` is the `type` you used in step 2.
- A service `type` must be unique within the DID document, see RFC006.
- To change the service: `PUT .../internal/didman/v1/did/<care organization DID>/compoundservice/<service type>`. To remove it: `DELETE .../internal/didman/v1/service/<service ID>`, with the `id` of the service in the DID document.
- The endpoints it refers to must already exist (step 2).
- An endpoint doesn't have to refer to your vendor DID: it can also be the URL itself, e.g. `"fhir": "https://fhir.example.com/base"`. You then have to change it in every care organization DID when the URL changes.

If your vendor DID has the compound service (step 2), refer to it as a whole:

```
POST <internal Nuts interface>/internal/didman/v1/did/<care organization DID>/endpoint
```

```
{
  "type": "<service type>",
  "endpoint": "<vendor DID>/serviceEndpoint?type=<service type>"
}
```

> **eOverdracht**

The sender adds an `eOverdracht-sender` service with an `oauth` and a `fhir` endpoint, see the eOverdracht leveranciersspecificatie:

```
{
  "type": "eOverdracht-sender",
  "serviceEndpoint": {
    "oauth": "<vendor DID>/serviceEndpoint?type=production-oauth",
    "fhir": "<vendor DID>/serviceEndpoint?type=eOverdracht-sender-fhir"
  }
}
```

The receiver adds an `eOverdracht-receiver` service with an `oauth` and a `notification` endpoint:

```
{
  "type": "eOverdracht-receiver",
  "serviceEndpoint": {
    "oauth": "<vendor DID>/serviceEndpoint?type=production-oauth",
    "notification": "<vendor DID>/serviceEndpoint?type=eOverdracht-receiver-notify"
  }
}
```

Find the endpoints of another care organization

To find care organizations by name that have the service of a use case:

```
GET <internal Nuts
interface>/internal/didman/v1/search/organizations?query=<name>&didServiceType=<service type>
```

- `query` is required. It matches part of the name, and names that sound like it.
- `didServiceType` is optional: the `<service type>` of step 4, e.g. `eOverdracht-receiver` to find care organizations that can receive an eOverdracht transfer.
- Each result has the organization's name and city, and its DID document. The DID is `didDocument.id`.
- A care organization is only returned when its DID document is active, and it has a NutsOrganizationCredential that is not expired or revoked, from an issuer you trust: see Register, find and authorize care organizations.

```
[
  {
```

```
    "didDocument": { "id": "did:nuts:...", "service": [ ... ] },
    "organization": { "name": "CareBears", "city": "CareTown" }
  }
]
```

Then resolve an endpoint of the service:

```
GET <internal Nuts interface>/internal/didman/v1/did/<DID>/compoundservice/<service
type>/endpoint/<endpoint name>
```

The node follows the reference to the vendor DID and returns the URL: `{"endpoint": "<URL>"}`, or plain text with the header `Accept: text/plain`.

API reference: DID Manager and Verifiable Data Registry (v1)

Register, find and authorize care organizations

Spec v1 uses two kinds of credentials:

Credential	Says	Issued by	Visibility
NutsOrganizationCredential	This DID is care organization X in city Y	The vendor, with its vendor DID	Public: every node gets it
NutsAuthorizationCredential (RFC014)	Organization B may access these resources of organization A	The care organization that holds the data (A), to the organization that may access it (B)	Private: only A's and B's nodes get it

Both are sent over the Nuts network (gRPC). The examples below use eOverdracht, see the eOverdracht leveranciersspecificatie.

> **Use case design (spec v1)**
>
> The use case specifies, per role (e.g. eOverdracht sender and receiver):
>
> - the `purposeOfUse` value of its NutsAuthorizationCredentials, and the access policy that belongs to it;
> - what the credentials contain, how long they are valid and when they are revoked.
>
> The node doesn't check these rules: an authorization credential that doesn't follow the use case gets access to nothing, or breaks interoperability.
>
> Trust works differently per credential: a NutsOrganizationCredential is only accepted from issuers that the node trusts. A NutsAuthorizationCredential is issued by the care organization itself, so it is never from a trusted issuer. See Credential Trust.

1. Register your care organizations

Issue a NutsOrganizationCredential for each care organization, with your vendor DID as issuer:

```
POST <internal Nuts interface>/internal/vcr/v2/issuer/vc
```

```
{
  "issuer": "<vendor DID>",
  "type": "NutsOrganizationCredential",
  "credentialSubject": {
    "id": "<care organization DID>",
    "organization": {
      "name": "<name>",
      "city": "<city>"
    }
  },
  "visibility": "public"
}
```

Issue all of them with the same vendor DID: other vendors have to trust every issuer DID separately.

- Required: `issuer`, `type` and `credentialSubject`. Optional: `expirationDate`, and `@context` (default: `https://nuts.nl/credentials/v1`).
- The node adds the `id` (`<issuer DID>#<UUID>`), `issuanceDate`, `@context` and `proof`, and returns the complete credential. You don't have to store it: you can find it again with a search.
- The combination of name and city should be unique: don't register two care organizations with the same name in the same city. The node doesn't check this.
- Any vendor can issue a NutsOrganizationCredential. That is why others must trust your vendor DID first, see step 2.

2. Trust other vendors

By default your node trusts no issuers, so a search finds no care organizations of other vendors. Trust each vendor's DID for NutsOrganizationCredential:

1. List the issuers you don't trust yet: `GET <internal Nuts interface>/internal/vcr/v2/verifier/NutsOrganizationCredential/untrusted`.
2. Check the DID with the other vendor, out-of-band, e.g. by phone. Ask them for their DID, rather than asking "is this your DID?". The contact information in a DID document isn't verified. It can help you find whom to call.
3. Trust it:

```
POST <internal Nuts interface>/internal/vcr/v2/verifier/trust
```

```
{
  "issuer": "<vendor DID>",
```

```
  "credentialType": "NutsOrganizationCredential"
}
```

You can also edit the file `vcr/trusted_issuers.yaml` in the data directory of your node (`datadir`).

3. Find the other care organization

Search for its NutsOrganizationCredential. The care organization's DID is in `credentialSubject.id`.

```
POST <internal Nuts interface>/internal/vcr/v2/search
```

```
{
  "query": {
    "@context": ["https://www.w3.org/2018/credentials/v1", "https://nuts.nl/credentials/v1"],
    "type": ["VerifiableCredential", "NutsOrganizationCredential"],
    "credentialSubject": {
      "organization": {
        "name": "<name>*"
      }
    }
  }
}
```

- `@context` and `type` are required, but don't filter: add at least one other field, or you get all credentials.
- By default a value must match exactly.
- A `*` at the end of a value matches everything that starts with it, e.g. `Hospital Amst*`.
- `"*"` alone matches every credential that has the field, with a value.
- A `*` anywhere else is not a wildcard, and wildcards only work on text.
- You can search on several fields at once, e.g. `credentialSubject.id`, `name` and `city`.
- The response is a list of complete credentials.

The endpoints of the organization are services in its DID document: see Set up DIDs and services for how to resolve them, and how to find care organizations that have the service of a use case.

> **eOverdracht**
>
> The sender looks up the `eOverdracht-receiver` service in the receiver's DID document. Its `notification` endpoint is where the sender sends the notification.

4. Authorize the other organization

The care organization that holds the data issues a NutsAuthorizationCredential to the organization that may access it. Issue it with the care organization's DID, as `private`: it can contain personal data.

```
POST <internal Nuts interface>/internal/vcr/v2/issuer/vc
```

```
{
  "issuer": "<your care organization DID>",
  "type": "NutsAuthorizationCredential",
  "credentialSubject": {
    "id": "<other care organization DID>",
    "purposeOfUse": "<purpose of use>",
    "resources": [ ... ]
  },
  "expirationDate": "<date>",
  "visibility": "private"
}
```

Field	Contains
`issuer`	The care organization that holds the data. Your node must manage its DID
`credentialSubject.id`	The care organization that gets access. Find its DID with a search, see step 3
`purposeOfUse`	Set by the use case. It selects the access policy
`resources`	What may be accessed. Per resource: `path`, `operations`, and `userContext`: whether a user identity is needed
`subject`	The patient, as `urn:oid:2.16.840.1.113883.2.4.6.3:<BSN>`. Optional: the use case says when to add it

See RFC014 for the fields. An example for one resource of one patient:

```
{
  "issuer": "<your care organization DID>",
  "type": "NutsAuthorizationCredential",
  "credentialSubject": {
    "id": "<other care organization DID>",
    "purposeOfUse": "<purpose of use>",
    "resources": [
```

```
      { "path": "/Patient/2250f7ab-6517-4923-ac00-88ed26f85843", "operations": ["read"],
"userContext": true }
    ],
    "subject": "urn:oid:2.16.840.1.113883.2.4.6.3:123456780"
  },
  "visibility": "private"
}
```

Private credentials are only exchanged between nodes that have proven who they are. So both your node and the other party's node need a node DID with a `NutsComm` endpoint, see Join the Nuts network.

eOverdracht

The sender issues two credentials to the receiver, both with `purposeOfUse` `eOverdracht-sender`: one for the Task (valid 1 year) and one for the patient data of the transfer. The leveranciersspecificatie lists their resources. The receiver needs no NutsAuthorizationCredential for its notification endpoint.

The Task credential:

```
{
  "issuer": "<sender DID>",
  "type": "NutsAuthorizationCredential",
  "credentialSubject": {
    "id": "<receiver DID>",
    "purposeOfUse": "eOverdracht-sender",
    "resources": [
      { "path": "/Task/<id>", "operations": ["read", "update"], "userContext": false }
    ]
  },
  "expirationDate": "<1 year from now>",
  "visibility": "private"
}
```

The patient data credential has the same fields, with these `resources`:

```
[
  { "path": "/Task/<id>", "operations": ["read", "update"], "userContext": false },
```

```
  { "path": "/Composition/<id>", "operations": ["read", "document"], "userContext": true
},
  { "path": "/<FHIR reference in the transfer>", "operations": ["read"], "userContext":
true }
]
```

Add a `resources` entry for each FHIR reference in the transfer. Revoke this credential when the Task is completed.

5. Use an authorization you received

Your node receives the credentials issued to your care organizations. Search for them with `allowUntrustedIssuer`: they're issued by care organizations, which you don't trust as issuers, so without it the search returns nothing.

```
POST <internal Nuts interface>/internal/vcr/v2/search
```

```
{
  "query": {
    "@context": ["https://www.w3.org/2018/credentials/v1", "https://nuts.nl/credentials/v1"],
    "type": ["VerifiableCredential", "NutsAuthorizationCredential"],
    "credentialSubject": {
      "id": "<your care organization DID>",
      "purposeOfUse": "<purpose of use>",
      "resources": {
        "path": "<path you want to call>"
      }
    }
  },
  "searchOptions": {
    "allowUntrustedIssuer": true
  }
}
```

The response is a list of complete credentials. The ID of a credential is its `id` field:

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

```
    "id": "<issuer DID>#314542e8-c8cc-4502-a7df-a815ac47c06b",
    "type": ["NutsAuthorizationCredential", "VerifiableCredential"],
    "issuer": "<issuer DID>",
    "issuanceDate": "2026-07-26T14:36:10.163463+02:00",
    "credentialSubject": { "id": "<your care organization DID>", "purposeOfUse": "...",
"resources": [ ... ] },
    "proof": { "type": "JsonWebSignature2020", ... }
  }
]
```

You can search on any field of the credential. E.g. to find all authorizations for one patient, search on `credentialSubject.id` and `credentialSubject.subject`:

```
"credentialSubject": {
  "id": "<your care organization DID>",
  "subject": "urn:oid:2.16.840.1.113883.2.4.6.3:123456780"
}
```

The `purposeOfUse` of each result says what you can request. This way the credentials also tell you where to find data of a patient.

Pass the credential in the access token request: see Request an access token.

> **eOverdracht**
>
> After the notification, the receiver searches with `purposeOfUse` `eOverdracht-sender` and `resources.path` `/Task/<id>`, and uses the credential to fetch the Task.

6. Revoke an authorization

Revoke a NutsAuthorizationCredential when access must end. The revocation is sent over the Nuts network: The credential ID is the `id` of the credential.

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

> **eOverdracht**
>
> When the Task is `completed`, the sender revokes the patient data credential. The Task credential stays valid until it expires.

API reference: Verifiable Credential Registry (v2)

Authenticate users

In spec v1, the user signs a *login contract*: a statement that they act on behalf of a care organization, valid for a limited time. The signed contract is a Verifiable Presentation. You send it as `identity` in an access token request, see Request an access token, so the other party knows which user requests the data. The contract identifies the user to other parties, and says for which care organization they work. You can also use the signed contract as the user's session token in your EHR.

> **Use case design (spec v1)**
>
> The use case specifies whether an access token request needs a user identity. Requests that return personal or medical data normally need one.

The user signs with the *employee identity* means (`employeeid`): the care organization states who its employee is (identifier, name, role), and the user confirms the contract in a browser. In the access token, `username` is the identifier you pass and `assurance_level` is `low`.

The node also supports IRMA (Yivi) as means, but it's no longer used and not described here.

Prerequisites

The user's browser talks to the Nuts node directly. So:

- the APIs of the node that start with `/public`, on its public interface (`:8080`), must be reachable over HTTPS from the internet, without extra access restrictions;
- `url` must be set to the URL they're reachable on:

```
url: https://<public domain>
```

v5: the setting is `auth.publicurl`.

How it fits in your application

The user's browser can't call the node's internal API. Your backend does that, and offers your frontend two endpoints:

Your backend endpoint	Does
Start signing	Draws up the contract (step 1), starts the session (step 2), returns `sessionPtr.url` and the session ID

Your backend endpoint	Does
Get the result	Asks the node for the session (step 3), returns the status. When signing is done, it stores the signed contract in the user's session

Your frontend shows the page at `sessionPtr.url`, then asks for the result until signing is done. Protect both endpoints with the user's EHR session. Pass errors on to the frontend, so the user knows what went wrong.

1. Draw up a contract

```
PUT <internal Nuts interface>/internal/auth/v1/contract/drawup
```

```
{
  "type": "PractitionerLogin",
  "language": "EN",
  "version": "v3",
  "legalEntity": "<care organization DID>",
  "validFrom": "2026-01-02T15:04:05+01:00",
  "validDuration": "2h"
}
```

- `type` / `language`: `BehandelaarLogin` with `NL`, or `PractitionerLogin` with `EN`.
- `version`: `v3`.
- `legalEntity`: the care organization the user acts for. Your node must manage its DID, and it needs a NutsOrganizationCredential, see Register, find and authorize care organizations: its name and city are in the contract text. The user picks the care organization at login, or is already logged in for it.
- `validFrom`: RFC3339 time, optional (default: now).
- `validDuration`: e.g. `2h` or `60m`, optional (default: 60 minutes).

If the care organization has NutsOrganizationCredentials with different names or cities, the request fails. Then pass the one to use as `organizationCredential`.

The response has the `type`, `language` and `version` of the contract, and the contract text in `message`. The dates in the text are in Dutch time (`Europe/Amsterdam`). E.g.:

```
EN:PractitionerLogin:v3 I hereby declare to act on behalf of CareBears located in CareTown.
This declaration is valid from Monday, 2 January 2026 15:04:05 until Monday, 2 January 2026
17:04:05.
```

2. Sign the contract

The user needs a recent browser with JavaScript. Start a signing session with the contract text as `payload`:

```
POST <internal Nuts interface>/internal/auth/v1/signature/session
```

```
{
  "means": "employeeid",
  "payload": "<message>",
  "params": {
    "employer": "<care organization DID>",
    "employee": {
      "identifier": "<unique user ID, e.g. email address or employee number>",
      "initials": "<initials>",
      "familyName": "<family name>",
      "roleName": "<role>"
    }
  }
}
```

All fields are required, except `roleName`: the user's role in the care organization. That is a job title, or the role you use for access control if people can read it.

The response:

```
{
  "sessionID": "<session ID>",
  "means": "employeeid",
  "sessionPtr": {
    "sessionID": "<session ID>",
    "url": "https://<public domain>/public/auth/v1/means/employeeid/<session ID>"
  }
}
```

Show the page at `sessionPtr.url` to the user (iframe, pop-up, or a browser window inside a native app), where they confirm the contract. The user has 10 minutes to do so. Make sure the user sees the page, e.g. bring it to the front. The page can be opened once: after a reload it no longer shows the contract. Then start a new session.

3. Get the signed contract

The endpoint is internal, so your frontend can't call it: it asks your backend. Poll the session from your backend:

```
GET <internal Nuts interface>/internal/auth/v1/signature/session/<sessionID>
```

```
{
  "status": "completed",
  "verifiablePresentation": { ... }
}
```

- `status`: `created`, `in-progress`, `completed`, `cancelled`, `errored` or `expired`. Use it to show progress to the user.
- `verifiablePresentation`: the signed contract. It's only there when signing succeeded. Store it in the user's session.

You get the signed contract once: only the first response with `completed` contains it. After that, the node removes the session: later calls don't return the contract and end in `404`. A cancelled, errored or expired session is also removed once you've read its status.

4. Check the signed contract before using it

The contract expires (`validDuration`). Check it before you send it:

```
PUT <internal Nuts interface>/internal/auth/v1/signature/verify
```

```
{
  "VerifiablePresentation": <signed contract>
}
```

Add `checkTime` (RFC3339) to check at another moment than now. `validity` is `false` when the contract is invalid or expired. Then let the user sign a new one.

The response for a valid contract:

```
{
  "validity": true,
  "vpType": "NutsSelfSignedPresentation",
  "issuerAttributes": {
    "username": "<identifier>",
    "initials": "<initials>",
    "familyname": "<family name>",
    "user_role": "<role>",
    "assurance_level": "low"
  },
```

```
  "credentials": {
    "legal_entity": "<organization name>",
    "legal_entity_city": "<organization city>",
    "valid_from": "<start, as in the contract text>",
    "valid_to": "<end, as in the contract text>"
  }
}
```

API reference: Auth (v1)

Request an access token

To call an endpoint of another care organization, your node requests an access token at that organization's node (Nuts RFC003). The other party's node checks the request and, if it's valid, issues the token. Its resource server then decides whether the token gives access to the requested data.

Role	Who	Does
Requester	Your care organization, through your node	Requests the token, calls the endpoint with it
Authorizer	The care organization that holds the data	Its node issues the token; its resource server checks it

If you're the authorizer, see Check an access token.

> **Use case design (spec v1)**
>
> The use case specifies, per endpoint:
>
> - the service the token is requested for (`service`, e.g. `eOverdracht-sender`);
> - which NutsAuthorizationCredentials must be included, and whether a user identity is needed.

Prerequisites

- The authorizer's DID document has the service, with an `oauth` endpoint that is the full URL of its node's token endpoint, e.g. `https://<node domain>/n2n/auth/v1/accesstoken` (HTTPS in strict mode): see Set up DIDs and services.
- The authorizer's node trusts your vendor DID as issuer of NutsOrganizationCredentials, and your care organization has one: see Register, find and authorize care organizations.

1. Request the access token

```
POST <internal Nuts interface>/internal/auth/v1/request-access-token
```

```
{
  "requester": "<your care organization DID>",
  "authorizer": "<other care organization DID>",
```

```
  "service": "<service type>",
  "credentials": [ <NutsAuthorizationCredential> ],
  "identity": <signed login contract>
}
```

- `credentials`: the full NutsAuthorizationCredentials the authorizer issued to you, as found with a search (see Register, find and authorize care organizations, step 5). Use `[]` if the use case needs none.
- `identity`: the signed login contract of the user, see Authenticate users. Leave it out if the use case doesn't need a user identity.

Your node:

1. resolves the `oauth` endpoint of the `service` in the authorizer's DID document;
2. creates a JWT with the request, signed with your care organization's key;
3. sends it to that endpoint, and returns the response.

```
{
  "access_token": "<access token>",
  "token_type": "bearer",
  "expires_in": 60
}
```

Errors:

- `400`: the request is invalid, e.g. the authorizer's DID document has no such service or `oauth` endpoint, or a credential is invalid.
- `503`: the authorizer's node couldn't be reached or refused the request; its log says why.

The JWT is valid for 5 seconds, so the clocks of both nodes must be in sync (allowed difference: `auth.clockskew`, default 5 seconds).

2. Call the endpoint

Send the token with each request to the authorizer's endpoint, until it expires. In strict mode a token is valid for 60 seconds (`expires_in`), so request a new one when needed:

```
Authorization: Bearer <access token>
```

Resolve the endpoint's URL as described in Set up DIDs and services. Whether the endpoint needs a TLS client certificate is set by the use case.

eOverdracht

Three token requests, see the eOverdracht leverancierssspecificatie:

Who, for what	`service`	`credentials`	`identity`
Sender, to notify the receiver	`eOverdracht-receiver`	none	no
Receiver, to fetch or update the Task	`eOverdracht-sender`	The Task credential	no
Receiver, to fetch the patient data	`eOverdracht-sender`	The patient data credential	yes

All calls use mTLS, with the same kind of certificate as the nodes.

API reference: Auth (v1)

Check an access token

When another care organization calls your endpoint, it first requests an access token at your node (Nuts RFC003), see Request an access token. Your node checks the request and, if it's valid, issues the token. Your resource server then decides whether the token gives access to the requested data.

> **Use case design (spec v1)**
>
> The use case specifies an *access policy* for each service: which requests a token for that service allows. A request can be allowed by the token alone, or only when a NutsAuthorizationCredential in the token lists the resource. Your resource server applies this policy; the node doesn't.

Prerequisites

- Your node accepts calls on `/n2n` from other nodes, on its public interface (`:8080`). Your proxy checks the TLS client certificate; the node doesn't do TLS itself: see Recommended deployment in the Nuts node documentation. **v5:** the node can also do this itself.
- Your care organization's DID document has the service of the use case, with an `oauth` endpoint that is the full URL of your node's token endpoint, e.g. `https://<node domain>/n2n/auth/v1/accesstoken`: see Set up DIDs and services.
- Your node trusts the requester's vendor DID as issuer of NutsOrganizationCredentials: see Register, find and authorize care organizations.

1. Your node issues the token

Your node issues the token only when all of these hold:

- The JWT has a valid signature and isn't expired.
- The requester has a NutsOrganizationCredential from an issuer your node trusts.
- Your node manages the authorizer DID: your care organization.
- The JWT is meant for the `oauth` endpoint of the `service` in the authorizer's DID document.
- If there's an `identity`: the contract is valid, and its organization name and city match a NutsOrganizationCredential of the requester.
- Each NutsAuthorizationCredential is valid, not revoked, issued by the authorizer and issued to the requester.

It doesn't check:

- which data the request is for, or whether the credentials fit the `service`: that's up to your resource server;
- credentials of other types: they're passed on to the token unchecked.

2. Check the token at your resource server

For each incoming request, introspect the token (RFC7662):

```
POST <internal Nuts interface>/internal/auth/v1/accesstoken/introspect
Content-Type: application/x-www-form-urlencoded

token=<access token>
```

Deny the request if `active` is `false`. Otherwise, check:

Field	Contains	Check
`service`	The service the token was requested for	Matches the endpoint that is called
`iss`	The authorizer DID	Is the care organization whose data is requested
`sub`	The requester DID	Record in your audit log. Decide whether this care organization gets access to the requested data. A NutsAuthorizationCredential issued to it is one basis, but you may need more, such as the patient's explicit consent or a treatment relationship. What you check depends on the use case and on what the credential was issued for
`resolvedVCs`	The credentials of the request	The `purposeOfUse` fits the `service`. The requested path and operation are in `resources` of one of them, or allowed by the use case's access policy. If that resource has `userContext: true`, the token has a user
`username`, `initials`, `prefix`, `family_name`, `email`, `user_role`, `assurance_level`	The user, if the request had an `identity`	Record in your audit log; `username` identifies the user

The response can have more fields than these.

`vcs` lists the IDs of all credentials in the request. `resolvedVCs` only has those that are valid now: a credential that was revoked or expired after the token was issued is left out. So use `resolvedVCs`, not `vcs`.

A reverse proxy in front of your resource server can check a token with `HEAD <internal Nuts interface>/internal/auth/v1/accesstoken/verify` and the token in the `Authorization` header: `200` means valid, `403` invalid. The proxy calls the node's internal interface itself; never make `/internal` reachable from outside. This call doesn't return the token's content, so your resource server still introspects it for the checks above.

> **eOverdracht**
>
> A token for `eOverdracht-receiver` only allows a notification. A token for `eOverdracht-sender` only allows the resources in its credentials.
>
> The sender gets the requests for the Task and the patient data (`service` `eOverdracht-sender`). The receiver gets the notification (`service` `eOverdracht-receiver`, without credentials or user). See the table in Request an access token.
>
> All calls use mTLS, with the same kind of certificate as the nodes.

API reference: Auth (v1)