

---

# Nuts Node Documentation

**Nuts community**

**Jun 01, 2022**



# GETTING STARTED:

1	Getting Started on Docker	1
2	Getting Started with native binary	3
3	Setting up your node for a network	5
4	Getting Started on customer integration	9
5	Signing a contract with IRMA	15
6	Getting started with Authorizations	23
7	Getting Started on EHR integration	29
8	Decentralized identifiers	31
9	Verifiable Credentials	33
10	VC Concept mapping	35
11	Nuts node development	39
12	API development	41
13	Issuing and searching Verifiable Credentials	43
14	Events	47
15	Nuts Node APIs	49
16	Nuts node config	51
17	Contribute	53
18	Release notes	55
19	Roadmap	57
20	Nuts node monitoring	59
21	Adminstering your node	65
22	Configuring for Production	67



## GETTING STARTED ON DOCKER

If you already use Docker, the easiest way to get your Nuts Node up and running for development or production is using Docker. To use the latest *master* build use *nutsfoundation/nuts-node:master* (for production environments it's advisable to use a specific version).

First determine the working directory for the Nuts node which will contain configuration and data. These which will be mounted into the Docker container. Follow the [configuration](#) to setup the configuration of your node.

### 1.1 Mounts

Using this guide the following resources are mounted:

- Readonly PEM file with TLS certificate and private key. They can be separate but in this example they're contained in 1 file.
- Readonly PEM file with TLS truststore for the particular network you're connecting to.
- Readonly *nuts.yaml* configuration file.
- Data directory where data is stored.

### 1.2 Docker run

If you want to run without Docker Compose you can use the following command from the working directory:

```
docker run --name nuts -p 5555:5555 -p 1323:1323 \
  --mount type=bind,source="$(pwd)"/certificate-and-key.pem,target=/opt/nuts/certificate-
↪and-key.pem,readonly \
  --mount type=bind,source="$(pwd)"/truststore.pem,target=/opt/nuts/truststore.pem,
↪readonly \
  --mount type=bind,source="$(pwd)"/nuts.yaml,target=/opt/nuts/nuts.yaml,readonly \
  --mount type=bind,source="$(pwd)"/data,target=/opt/nuts/data \
  -e NUTS_CONFIGFILE=/opt/nuts/nuts.yaml \
  nutsfoundation/nuts-node:master
```

This setup uses the following *nuts.yaml* configuration file:

```
datadir: /opt/nuts
network:
  truststorefile: /opt/nuts/truststore.pem
  certfile: /opt/nuts/certificate-and-key.pem
```

(continues on next page)

(continued from previous page)

```
certkeyfile: /opt/nuts/certificate-and-key.pem
bootstrapnodes:
  - example.com:5555
```

---

**Note:** The command above uses *pwd* and *bash* functions, which do not work on Windows. If running on Windows replace it with the path of the working directory.

---

You can test whether your Nuts Node is running properly by visiting <http://localhost:1323/status/diagnostics>. It should display diagnostic information about the state of the node.

## 1.3 Docker Compose

Copy the following YAML file and save it as *docker-compose.yaml* in the working directory.

```
version: "3.7"
services:
  nuts:
    image: nutsfoundation/nuts-node:master
    environment:
      NUTS_CONFIGFILE: /opt/nuts/nuts.yaml
    ports:
      - 5555:5555
      - 1323:1323
    volumes:
      - "./certificate-and-key.pem:/opt/nuts/certificate-and-key.pem:ro"
      - "./truststore.pem:/opt/nuts/truststore.pem:ro"
      - "./nuts.yaml:/opt/nuts/nuts.yaml:ro"
      - "./data:/opt/nuts/data:rw"
```

Start the service:

```
docker-compose up
```

## GETTING STARTED WITH NATIVE BINARY

The Nuts executable this project provides can be used to both run a Nuts server (a.k.a. node) and administer a running node remotely. This chapter explains how to run a server using the native binary.

### 2.1 Building

Since no precompiled binaries exist (yet), you'll have to build the binary for your platform.

First check out the project using:

```
git clone https://github.com/nuts-foundation/nuts-node
cd nuts-node
```

Then create the executable using the *make* command:

```
make build
```

Or if make is not available:

```
go build -ldflags="-w -s -X 'github.com/nuts-foundation/nuts-node/core.GitCommit=GIT_
↪COMMIT' -X 'github.com/nuts-foundation/nuts-node/core.GitBranch=GIT_BRANCH' -X 'github.
↪com/nuts-foundation/nuts-node/core.GitVersion=GIT_VERSION'" -o /path/to/nuts
```

Make sure *GIT\_COMMIT*, *GIT\_BRANCH* and *GIT\_VERSION* are set as environment variables. These variables help identifying an administrator and other nodes what version your node is using. If this isn't important then replace *GIT\_COMMIT* with *0*, *GIT\_BRANCH* with *master* and *GIT\_VERSION* with *undefined*.

### 2.2 Starting

Start the server using the *server* command:

```
nuts server
```

Now continue with the *configuration*.





## SETTING UP YOUR NODE FOR A NETWORK

After you managed to start your node using either *docker* or *native* it's time to connect to a network.

### 3.1 Prerequisites

The following is needed to connect a Nuts node to a network:

1. A runnable node.
2. The public address of one or more remote nodes you'd like to use as bootstrap nodes.
3. A TLS client- and server certificate which is accepted by the other nodes in the network (e.g. PKIoverheid).
4. A truststore containing the CA trust anchors for TLS certificates the network you're connecting to accepts (e.g. PKIoverheid).

To connect to the development network you can use the `nuts-development-network-ca` by cloning <https://github.com/nuts-foundation/nuts-development-network-ca>. That project contains scripts to generate certificates, and a truststore.

### 3.2 Configuring

1. Configure the bootstrap nodes using `network.bootstrapnodes`.
2. Configure TLS using `network.certfile`, `network.certkeyfile` and `network.truststorefile`.

See *configuration reference* for a detailed explanation on how to exactly configure the Nuts node.

---

**Note:** You *can* start the node without configuring the network, but it won't connect and thus exchange data with other nodes.

---

### 3.2.1 YAML Configuration File

If you're using a YAML file to configure your node, the following snippet shows an example for the network related configuration:

```
network:
  truststorefile: /path/to/truststore.pem
  certfile: /path/to/certificate-and-key.pem
  certkeyfile: /path/to/certificate-and-key.pem
  bootstrapnodes:
    - example.com:5555
```

### 3.2.2 Node TLS Certificate

To connect to an existing Nuts network you need a TLS certificate which authenticates your node. For the development network you can use the `nuts-network-development-ca` to directly issue a certificate for your node. The commands below clone the required Git repository, generate a private key and issues a certificate, and combines them into a single file:

```
git clone https://github.com/nuts-foundation/nuts-development-network-ca
cd nuts-development-network-ca && ./issue-cert.sh localhost
cat localhost.key localhost.pem > certificate-and-key.pem
```

---

**Note:** If you want peers to be able to connect to your node, replace `localhost` with the correct hostname.

---

Note that the Git repository contains the Certificate Authority certificate (`ca.pem`) which will function as `truststore`. Copy this file as `truststore.pem` into the working directory.

### 3.2.3 Node Identity

Certain data (e.g. private credentials) can only be exchanged when a peer's DID has been authenticated. To make sure other nodes can authenticate your node's DID you need to configure your node's identity, and make sure the DID document contains a `NutsComm` service that matches the TLS certificate.

Your node identity is expressed by a DID that is managed by your node, also known as your *vendor DID*. So make sure you have created a DID specific for your nodes and configure it as `network.nodedid` (see [configuration reference](#)).

Then you make sure the associated DID Document contains a `NutsComm` endpoint, where the domain part (e.g. `nuts.nl`) matches (one of) the DNS SANs in your node's TLS certificate. See "Node Discovery" below for more information on registering the `NutsComm` endpoint.

---

**Note:** Multiple nodes may share the same DID, if they're governed by the same organization (e.g., clustered setups).

---

### 3.2.4 Node Discovery

To allow your Nuts node to be discovered by other nodes, so they can connect to it, you need to register a `NutsComm` endpoint on your vendor DID document. The `NutsComm` endpoint contains a URL to your node's public gRPC service, and must be in the form of `grpc://<host>:<port>`. E.g., if it were to run on `nuts.nl:5555`, the value of the `NutsComm` endpoint should be `grpc://nuts.nl:5555`

You can register the `NutsComm` endpoint by calling `addEndpoint` on the DIDMan API:

```
POST <internal-node-address>/internal/didman/v1/did/<vendor-did>/endpoint
{
  "type": "NutsComm",
  "endpoint": "grpc://nuts.nl:5555"
}
```

## 3.3 Care Organizations

The DID documents of your care organizations you (as a vendor) want to expose on the Nuts network need to be associated with your vendor's DID document through the `NutsComm` endpoint. It's recommended to register the actual `NutsComm` endpoint on your vendor DID document (as explained in the previous section), and register a reference to this endpoint on the DID documents of your vendor's care organizations:

```
POST <internal-node-address>/internal/didman/v1/did/<care-organization-did>/endpoint
{
  "type": "NutsComm",
  "endpoint": "<vendor-did>/serviceEndpoint?type=NutsComm"
}
```



## GETTING STARTED ON CUSTOMER INTEGRATION

This getting started manual assumes a vendor sells services to its customers. The vendor manages the presence of those customers on the Nuts network through the Nuts registry. It's very likely the vendor has software to manage customer environments. We'll call it a *CRM* where the *customers* part relates to organizations and not people. This does not mean that a stand-alone installation isn't supported. In that case the vendor and organization are the same.

The Nuts registry enables service discovery for organizations. The registry identifies organizations through their *DID*. The DIDs are unique identifiers which are generated when the organization is registered in the Nuts registry. After creation of the DID the CRM should store and map it to its customer record, so it can refer to it when updating the customer's DID Document and issue Verifiable Credentials.

All APIs used in the following chapters are documented at the [API](#) page. Open API Spec files are available for generating client code.

### 4.1 Vendor integration

As a vendor, you're in power of:

- running a Nuts node
- handling organization key material
- updating the organization DID Document
- defining service endpoints
- issuing name credentials for organizations
- trusting other vendors

The last three points require a setup where a vendor DID is created. This DID will act as the controller of all organization DID Documents. This will allow for reuse of service endpoints and issuance of Verifiable Credentials. Since every DID issuing Verifiable Credentials must be trusted individually, it's easier for other vendors when the vendor uses a single DID for issuing credentials.

### 4.1.1 Create and store a vendor DID

Your CRM must store the DIDs created for your vendor and your customers. A DID is a string similar to:

```
did:nuts:2mF6KT6eiSx5y2fwTP4Y42yMUh91zGVkbu4KMARvCJz9
```

The DID we're about to create is your *vendor DID*. It will be used in all of the next steps. For the API calls that will need to be made to the Nuts node, we'll use `<internal-node-address>` as the address where the internal API's are exposed. Consult the [configuration reference](#) on how to configure the node address.

```
POST <internal-node-address>/internal/vdr/v1/did
{
  "selfControl": true,
  "keyAgreement": true,
  "assertionMethod": true,
  "capabilityInvocation": true
}
```

The request above instructs the node to create a new DID and DID Document. The DID Document will be published to all other nodes. The node will generate a new keypair and store it in the crypto backend. The options above will instruct the node to allow the DID Document to be changed by itself (`selfControl = true` AND `capabilityInvocation = true`) and that the DID can be used to issue credentials (`assertionMethod = true`). If all is well, the node will respond with a DID Document similar to:

```
{
  "@context": [ "https://www.w3.org/ns/did/v1" ],
  "id": "did:nuts:2mF6KT6eiSx5y2fwTP4Y42yMUh91zGVkbu4KMARvCJz9",
  "verificationMethod": [
    {
      "id": "did:nuts:2mF6KT6eiSx5y2fwTP4Y42yMUh91zGVkbu4KMARvCJz9#_
      ↪TKzHv2jFIyvdTGF1Dsgwngfdg3SH6TpDv0Ta1a0Ekw",
      "controller": "did:nuts:2mF6KT6eiSx5y2fwTP4Y42yMUh91zGVkbu4KMARvCJz9",
      "type": "JsonWebKey2020",
      "publicKeyJwk": {
        "crv": "P-256",
        "x": "38M1FDts7Oea7urmseiugGW7tWc3mLpJh6rKe7xINZ8",
        "y": "nDQW6XZ7b_u2Sy9slofYLLG03s0Eoug3I0aAPQ0exs4",
        "kty": "EC"
      }
    }
  ],
  "capabilityInvocation": [
    "did:nuts:2mF6KT6eiSx5y2fwTP4Y42yMUh91zGVkbu4KMARvCJz9#_
    ↪TKzHv2jFIyvdTGF1Dsgwngfdg3SH6TpDv0Ta1a0Ekw"
  ],
  "assertion": [
    "did:nuts:2mF6KT6eiSx5y2fwTP4Y42yMUh91zGVkbu4KMARvCJz9#_
    ↪TKzHv2jFIyvdTGF1Dsgwngfdg3SH6TpDv0Ta1a0Ekw"
  ],
  "keyAgreement": [
    "did:nuts:2mF6KT6eiSx5y2fwTP4Y42yMUh91zGVkbu4KMARvCJz9#_
    ↪TKzHv2jFIyvdTGF1Dsgwngfdg3SH6TpDv0Ta1a0Ekw"
  ],
  "service": []
}
```

The id at the top level needs to be extracted and stored as your vendor DID. In the example above this would be `did:nuts:2mF6KT6eiSx5y2fwTP4Y42yMUh91zGVkbu4KMARvCJz9`. The DID Document shouldn't be stored since the Nuts node will do this for you.

### 4.1.2 Setting vendor contact information

Things can go wrong: a node is misbehaving or a DID Document is conflicted. If the node operator is not resolving the problem it's extremely convenient if others can contact the node operator and relay the problem. For this use-case, Nuts supports the registration of node contact information. The contact information will be added to a DID Document as a service. A convenience API is available to add the contact information to a DID Document. The vendor DID should be used for this.

```
PUT <internal-node-address>/internal/didman/v1/did/<did>/contactinfo
{
  "name": "vendor X",
  "phone": "06-12345678",
  "email": "info@example.com",
  "website": "https://example.com"
}
```

Where `<did>` must be replaced with the vendor DID.

### 4.1.3 Adding endpoints

As a vendor you'll probably be hosting different services at various stages. A Nuts node API is available to easily add/remove the endpoints for these services. Registering services is a required step since the services that will be registered for organizations will make use of these services.

```
POST <internal-node-address>/internal/didman/v1/did/<did>/endpoint
{
  "type": "example-production-api",
  "endpoint": "https://api.example.com"
}
```

Where `<did>` must be replaced with the vendor DID. The `type` may be freely chosen and is used as reference in the organization services. The `endpoint` must be a valid endpoint (this differs per type of service). For some services this could be a base-url. If this is the case, the bolt description will note this.

## 4.2 Organization integration

Each organization (or customer) must be registered with its own DID and DID Document. The vendor CRM should make it possible to store a DID for each organization. Requests that are made in the context of the organization will use the private key of the organization. To easily control the DID Document of an organization, the vendor will be the controller.

### 4.2.1 Create and store a customer DID

A DID can be created like the vendor DID:

```
POST <internal-node-address>/internal/vdr/v1/did
{
  "selfControl": false,
  "controllers": [<did>],
  "assertionMethod": true,
  "capabilityInvocation": false
}
```

Where <did> must be replaced with the vendor DID. The body for creating an organization DID differs from the vendor DID in the fact that the vendor DID is in control of the newly generated DID Document. The `assertionMethod` is still true since it'll allow for the generation of access-tokens in the context of the organization. The result is similar to the output of the vendor DID creation. In this case the `id` must also be extracted and stored within the vendor CRM for the right organization.

### 4.2.2 Issue a Nuts Organization Credential

After registering an organization, its presence on the network and in the Nuts registry is now only a DID. In order for other organizations to find the correct DID and connected services, credentials should be issued and published over the network. For this, the *NutsOrganizationCredential* can be issued by any vendor. A *NutsOrganizationCredential* contains the name of the organization and the city where this name is registered as organization. The combination of those should be unique (since duplicate names within a sector is disallowed).

A credential can be issued with the following call:

```
POST <internal-node-address>/internal/vcr/v2/issuer/vc
{
  "type": "NutsOrganizationCredential",
  "issuer": "<issuer-did>",
  "credentialSubject": {
    "id": "<holder-did>",
    "organization": {
      "name": "<name>",
      "city": "<city>"
    }
  },
  "visibility": "public"
}
```

Where <issuer-did> must be replaced with the vendor DID, <holder-did> must be replaced with the organization DID, ``<name>`` and <city> must be replaced with the correct information. The API will respond with the full Verifiable Credential. It's not required to do anything with that since issued credentials can be found again.



### 4.2.3 Trusting other vendors as issuer

A node operator must not blindly trust all the data is published over the network. Before credentials can be found, the issuer has to be trusted. By default, no issuers are trusted. A list of untrusted issuers can be obtained from the node through:

```
GET <internal-node-address>/internal/vcr/v2/verifier/NutsOrganizationCredential/untrusted
```

This will return a list of all DIDs that are currently not trusted. If a DID is to be trusted should be validated out-of-band, eg: by phone or video conference call. The registered contact information for that DID could help in contacting the right party. Be aware that the provided contact information isn't verified. So instead of asking: "is this your DID?", ask: "could you please tell me your DID?". After a DID has been verified, it can be trusted by calling the following API:

```
POST <internal-node-address>/internal/vcr/v2/verifier/trust
{
  "issuer": "<did>",
  "credentialType": "NutsOrganizationCredential"
}
```

Where <did> must be replaced with the validated DID. It's also possible to update the `vcr/trusted_issuers.yaml` file located in the data directory (configured via the `datadir` property). After a vendor has been trusted, any of its registered organizations should be searchable by name.

**Note:** Future development will see new cryptographic means. These means could enable the organization to self-register its name. The network should then migrate to a trust model where the issuer of those means is trusted instead of the different vendors.

### 4.2.4 Enabling a bolt

Organizations can be found on the network and endpoints have been defined. Now it's time to enable specific bolts so users can start using data from other organizations. Every bolt requires its own configuration. This configuration is known as a Compound Service on the organization's DID document. A Compound Service defines certain endpoint types and which endpoint to use for that type.

A Compound Service can be added with the following request:

```
POST <internal-node-address>/internal/didman/v1/did/<did>/compoundservice
{
  "type": "<type>",
  "serviceEndpoint": {
    "<X>": "<endpoint_did>/serviceEndpoint?type=<Y>",
    ...
  }
}
```

The parameters must be replaced:

- <did> must be replaced with the organization DID.
- <type> must be replaced with the type defined by the bolt specification.
- <endpoint\_did> must be replaced with the vendor DID that defines the endpoints.

- <X> must be replaced with the type required by the bolt specification. All types defined by the specification must be added, unless stated otherwise.
- <Y> must be replaced with the correct endpoint type from the vendor DID Document. <endpoint\_did>/serviceEndpoint?type=<Y> must be a valid query within the corresponding DID Document.

For example, the [eOverdracht sender](#) requires an `eOverdracht-sender` Compound Service with two endpoints: an `oauth` endpoint and a `fhir` endpoint. The example can be added by the following request:

```
POST <internal-node-address>/internal/didman/v1/did/did:nuts:organization_identifier/  
↪compoundservice  
{  
  "type": "eOverdracht-sender",  
  "serviceEndpoint": {  
    "oauth": "did:nuts:vendor_identifier/serviceEndpoint?type=production-oauth",  
    "fhir": "did:nuts:vendor_identifier/serviceEndpoint?type=eOverdracht-sender-fhir"  
  }  
}
```

---

**Note:** As specified by [RFC006](#), the type MUST be unique within a DID Document.

---

## **SIGNING A CONTRACT WITH IRMA**

This getting started manual shows how to successfully use IRMA to sign a contract. Contracts are used within the Nuts ecosystem to identify a user to other network participants. It also relates a user to the care organization that user is currently working for. The signed contract is used as token to authenticate the user's (local) EHR identity to other nodes in the network and can be used as session token on the EHR. The contract is required for every request that results in personal and/or medical data being retrieved.

### **5.1 Basic requirements**

To use IRMA as a means for signing a contract, the following is required:

- the user has the IRMA app installed on an Android or iOS device with camera and an internet connection.
- the user has retrieved the BRP and email credentials in the IRMA app.
- the user interacts with the XIS/ECD via a recent browser capable of running javascript.
- the vendor has a Nuts node running.

### **5.2 IRMA flow**

We use the Nuts node as IRMA server and as tool to start an IRMA session. This follow the flow as described on this [IRMA Github page](#). The XIS/ECD will have to provide two endpoints for the frontend. One endpoint to start a session and one to get the session result. More info on these endpoints will be provided further down.

### **5.3 Configuring the Nuts node**

In the contract signing flow, the device running the IRMA app communicates with the Nuts node directly. Therefore the Nuts node needs to be accessible to the public internet. All APIs on the Nuts node starting with `/public` (without a trailing slash) must be accessible over HTTPS without any additional security measures that could prevent access by mobile devices. A domain must also be available which resolves to those APIs. The domain must be configured on the Nuts node:

**auth:**  
**publicurl:** `https://example.com`

The Nuts APIs used for signing will embed this URL in the QR code shown to the user. The javascript in the frontend will also use this URL (exposed via the QR code) to check the status of the signing session. Therefore the domain

which serves the frontend must be able to do requests to that domain. The browser will require CORS headers to be configured on the domain configured in the Nuts node config. This can be done by the following snippet:

```
http:
  default:
    cors:
      origin: "other.com"
```

Where *other.com* is the domain serving the frontend. For development purposes *\** is also allowed. If the public APIs are mounted on a different port/interface in the nuts config then the `default` key should be changed to `public` in the example above.

## 5.4 Setting up the frontend

For the frontend we'll be using the `irma-frontend-packages` javascript library. More info on how to use this library can be found on [`https://irma.app/docs/irma-frontend/`](https://irma.app/docs/irma-frontend/). You can choose to load the IRMA frontend packages javascript via an HTML tag, in which case you'll need to build the javascript file yourself given the instructions on [`https://github.com/privacybydesign/irma-frontend-packages`](https://github.com/privacybydesign/irma-frontend-packages) or you can choose to use npm:

```
"dependencies": {
  "@privacybydesign/irma-frontend": "^0.3.3"
}
```

Make sure you use the latest version.

IRMA allows for multiple frontends to be used. The most important ones are the *web* and *popup* frontends. The *web* frontend allows for embedding the IRMA web component within a html element. The *popup* frontend will render a new component that will render on top of the rest of the website. This manual will use the *popup* frontend.

A complete example:

```
let options = {
  // Developer options
  debugging: true,

  // Front-end options
  language: 'en',

  // customize textual components
  translations: {
    header: "Sign your contract"
  },

  // Back-end options
  session: {
    // Point to your web backend
    url: '/web/auth',

    // The request that will be send to the backend:
    start: {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json'
      }
    }
  }
}
```

(continues on next page)

(continued from previous page)

```

    },
    body: JSON.stringify(this.some_data)
  },

  // required to translate Nuts specific return values
  mapping: {
    sessionPtr:      r => r.sessionPtr.clientPtr,
    sessionToken:    r => r.sessionID
  }
}
};

// we'll use the popup frontend
let irmaPopup = irma.newPopup(options);

// start the interaction
irmaPopup.start()
  .then(result => {
    console.log("success!")
    console.log(response)
  })
  .catch(error => {
    if (error === 'Aborted') {
      console.log('Aborted');
      return;
    }
    console.error("error", error);
  })
  .finally(() => irmaPopup = irma.newPopup(options));
}

```

Lets break this down into parts.

```

// Developer options
debugging: true,

```

Is used to enabling debugging. The IRMA library will output more information helpful for development.

```

// Front-end options
language: 'en',

// customize textual components
translations: {
  header: "Sign your contract"
},

```

Sets the language to english which will set some default textual representations on the IRMA web component. The translations configuration option can be used to change each of the textual representation on the IRMA web component. In this case, only the header is changed.

```

// Back-end options
session: {
  // Point to your web backend

```

(continues on next page)

(continued from previous page)

```
url: '/web/auth',

// The request that will be send to the backend:
start: {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(this.some_data)
},

// required to translate Nuts specific return values
mapping: {
  sessionPtr:      r => r.sessionPtr.clientPtr,
  sessionToken:    r => r.sessionID
}
}
```

The `session` object contains all the technical parts to connect the IRMA javascript library to your backend. The contents of the `start` object configures the initial request to start a signing session. You can control the type of request and the contents. In this case, some data from the frontend is sent as JSON. This is optional and no particular data is required. The `url`, in this case `/web/auth`, must be set so the frontend can access the following URLs:

```
<url>/session
<url>/session/<sessionToken>/result
```

These URLs must both be available on the backend. For the example above this means that both `/web/auth/session/` and `/web/auth/session/<sessionToken>/result` are available. The `<sessionToken>` is the token that will be returned by the call to `<url>/session/`. How to parse the result of that call and extract the token is done via the `mapping` object.

The `mapping` object is a map where two keys are expected: `sessionPtr` and `sessionToken`. `sessionPtr` must point to the data that is used to render the QR code. `sessionToken` must point to the session token used to get the result.

## 5.5 Setting up the backend

As discussed in the previous chapter, the backend is required to expose two APIs to the frontend:

```
<url>/session
<url>/session/<sessionToken>/result
```

No particular security context is required, you may require a user session if needed.

### 5.5.1 Starting a session

The `<url>/session` API is used to start a session. To start a session at the Nuts node, a valid contract has to be drawn up first. You can create such a contract with the following API on the Nuts node:

```
PUT /internal/auth/v1/contract/drawup
```

With the following body:

```
{
  "type": "BehandelaarLogin",
  "language": "NL",
  "version": "v3",
  "legalEntity": "did:nuts:90348275fjasihnva4857qp39hn",
  "validFrom": "2006-01-02T15:04:05+02:00",
  "validDuration": "2h"
}
```

The `type` must be one of the valid Nuts contract types, currently only `BehandelaarLogin` for Dutch and `PractitionerLogin` for English are supported. The `language` selects the correct language, `NL` for Dutch and `EN` for english. The `version` must be `v3`. The `legalEntity` must refer to the DID of the current organization. The user either selects an organization to login for, or is already logged in. The organization must have a DID as described in *Getting Started on customer integration*. `validFrom` is a RFC3339 compliant time string. `validDuration` describes how long the contract is valid for. Time unit strings are used like `1h` or `60m`, the valid time units are: “ns”, “us” (or “µs”), “ms”, “s”, “m”, “h”. The local system timezone is used to format the date and time string.

The return value looks like:

```
{
  "type": "PractitionerLogin",
  "language": "EN",
  "version": "v3",
  "message": "EN:PractitionerLogin:v3 I hereby declare to act on behalf of CareBears,
  ↳ located in CareTown. This declaration is valid from Monday, 2 January 2006 15:04:05,
  ↳ until Monday, 2 January 2006 17:04:05."
}
```

The message from this result is used in the next part. Start an IRMA session by calling the following API on the Nuts node:

```
POST /internal/auth/v1/signature/session
```

The body for this call looks like:

```
{
  "means": "irma",
  "payload": "<message>"
}
```

Where `message` is the result from the contract call. The result from this call must be passed directly to the frontend. If any transformation is done, the mapping setting in the frontend must be changed accordingly.

### 5.5.2 Getting the session result

The IRMA javascript frontend library will check for the status of the signing session. When the session has been completed it'll call the following url:

```
GET <url>/session/<sessionToken>/result
```

where <url> is the base url configured under `session.url` in the javascript options and <sessionToken> is the token returned by the previous call. The backend must implement this API, the implementation must call the following API on the Nuts node:

```
GET /internal/auth/v1/signature/session/<sessionToken>
```

Any error in calling this service need to be relayed to the frontend. This will instruct the user on why things went wrong and what to do next. The call to the Nuts node will return the following response:

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

The `status` field has a different content when a different signing means is used. The presence of the `verifiablePresentation` in the result is the main method of checking if the signing session succeeded. `verifiablePresentation` is the cryptographic proof that needs to be stored in the user session. It's required in the OAuth flow for obtaining an access token. The backend should check if the signed contract (verifiable presentation) is still valid when using it. The validity can be checked by calling the following API with the verifiable presentation at the place of <vp>:

```
PUT /internal/auth/v1/signature/verify
```

with

```
{
  "checkTime": "2006-01-02T15:54:05+02:00",
  "verifiablePresentation": <vp>
}
```

It will return a structure similar to:

```
{
  "validity": true,
  "vpType": "NutsIrmaPresentation",
  "issuerAttributes": {
    "pbdg.gemeente.personalData.initials": "T",
    "pbdg.gemeente.personalData.prefix": "",
    "pbdg.gemeente.personalData.familyname": "Tester",
    "pbdg.sidn-pbdg.email.email": "tester@example.com"
  },
  "credentials": {
    "organization": "CareBears",
    "validFrom": "2006-01-02T15:04:05+02:00",
    "validTo": "2006-01-02T17:04:05+02:00"
  }
}
```

(continues on next page)



(continued from previous page)

```
}  
}
```

The `validity` will indicate its validity. An expired contract is considered invalid.



## GETTING STARTED WITH AUTHORIZATIONS

Authorization is one of the three core concepts of Nuts (the others being identification and addressing).

### 6.1 Introduction

Authorization comes in the form of a `NutsAuthorizationCredential`. An authorization credential is a privately distributed credential that answers:

- which **custodian** controls the resources
- to which **patient** do the resources belong to
- which **actor** may access the resources
- what is the **scope** of the authorization
- on what **legal base** is the authorization provided
- which individual **resources** may be accessed

Authorization credentials are issued by the same party that will also control the access to the resources.

#### 6.1.1 Bolt

A Bolt is a functional and technical specification that translates a care process to technical requirements. An authorization is created for a particular Bolt. A Bolt specifies what the possible values of *purposeOfUse* can be. Each value corresponds to an access policy defined by the Bolt. Creating an authorization credential that is not according to a Bolt specification will have little effect or will even hinder interoperability. Particular requirements for a Bolt are not validated by the node, the node will only do the validations as specified.

#### 6.1.2 Prerequisites

Since authorization credentials are privately distributed, their exchange only happens over authenticated connections. To query authorization credentials (issued to your care organization(s)) or let the authorized party query an authorization credential you issued, you need to configure your node's identity and `NutsComm` endpoint. See [setting up your node for a network](#) for how to achieve this.

## 6.2 Registering a NutsAuthorizationCredential

Issuing an authorization credential is similar to issuing an organization credential. Both use the same API. New credentials will automatically receive an *id*, *issuanceDate*, *context* and *proof*. A DID requires a valid [assertionMethod](#) key.

The credential can be issued with the following call:

```
POST <internal-node-address>/internal/vcr/v2/issuer/vc
{
  "issuer": "did:nuts:JCJEi3waNGNhkmwVvFB3wdUsmDYPnTcZxYiWThZqgWKv",
  "type": "NutsAuthorizationCredential",
  "credentialSubject": {
    "id": "did:nuts:JCJEi3waNGNhkmwVvFB3wdUsmDYPnTcZxYiWThZqgWKv",
    "legalBase": {
      "consentType": "implied"
    },
    "resources": [
      {
        "path": "/patient/2250f7ab-6517-4923-ac00-88ed26f85843",
        "operations": ["read"],
        "userContext": true
      }
    ],
    "purposeOfUse": "test-service",
    "subject": "urn:oid:2.16.840.1.113883.2.4.6.3:123456780"
  },
  "visibility": "private"
}
```

As you can see, there are quite some fields to fill out. The following paragraphs will dig deeper into the different parts.

### 6.2.1 issuer

The *issuer* is the resource owner. It must be a DID of an organization for which you control the private key. The DID typically comes from your own administration, see also [:ref: Getting Started on customer integration <connecting-crm>`\\_](#).

### 6.2.2 type

The *type* must equal [*NutsAuthorizationCredential*], no exceptions.

### 6.2.3 visibility

VCs that are published to the network can be published publicly or private. When published private, only the issuer and subject can read the contents of the VC. VCs that contain personal information must be published privately (*visibility = private*). When the VC is to be read by anyone on the network, it should be published publicly (*visibility = public*).

### 6.2.4 credentialSubject.id

The *credentialSubject.id* is the receiver or *holder* of the credential. It must be a DID of an organization. This DID is typically found via a search call. The following call will search for an organization with the name *CareBears*.

```
POST <internal-node-address>/internal/vcr/v2/search
{
  "query": {
    "@context": [
      "https://www.w3.org/2018/credentials/v1",
      "https://nuts.nl/credentials/v1"
    ],
    "type": ["VerifiableCredential", "NutsOrganizationCredential"],
    "credentialSubject": {
      "organization": {
        "name": "CareBears"
      }
    }
  }
}
```

The [:ref:`VC manual <using-vcs>`\\_](#) contains some more information on how to perform searches.

### 6.2.5 credentialSubject.purposeOfUse

The *credentialSubject.purposeOfUse* field will be filled with a fixed value. A Bolt specification will describe what value to put here.

### 6.2.6 credentialSubject.subject

The *credentialSubject.subject* field identifies the patient. Resources that are scoped to a patient will have an authorization record with a patient identifier. It's possible for authorization records to not include this field. A Bolt specification should describe when to use this field and when not. The contents in this example is a **urn** with a Dutch citizens number.

### 6.2.7 credentialSubject.legalBase

This field describes the legal base from which the authorization credential originates. A Bolt will what values are to be entered.

### 6.2.8 credentialSubject.resources

The resources array describes what resources may be accessed with the authorization credential. Unless stated otherwise by the Bolt, these resources are in addition to any common resources listed by the access policy of the Bolt. A resource has 3 members: *path*, *operations* and *userContext*. See [the Nuts specification](#) for more detail.

## 6.3 Searching for authorization credentials

Authorization credentials can be used as a distributed index: *where can I find information for patient X?*. When an access token is requested via the API, references to the relevant authorization credentials are required.

To find the relevant authorization credentials, the credential search API can be used. To find all authorization credentials of a single patient:

```
POST <internal-node-address>/internal/vcr/v2/search
{
  "query": {
    "@context": [
      "https://www.w3.org/2018/credentials/v1",
      "https://nuts.nl/credentials/v1"
    ],
    "type": ["VerifiableCredential", "NutsAuthorizationCredential"],
    "credentialSubject": {
      "id": "did:nuts:JCJEi3waNGNhkmwVvFB3wdUsmDYPnTcZxYiWThZqgWKv",
      "subject": "urn:oid:2.16.840.1.113883.2.4.6.3:123456780"
    }
  },
  "searchOptions": {
    "allowUntrustedIssuer": true
  }
}
```

The call above includes a query for a particular *receiver* via the *credentialSubject.id* key. This would typically be a DID from your own administration. The second parameter defines the patient. This example will return a list of authorization credentials where the *credentialSubject.purposeOfUse* field will indicate what kind of information can be retrieved. The *untrusted* query parameter must be added because authorization credentials are not issued by a trusted third party but by organizations themselves.

It can also be the case that you need to find an authorization that covers a certain request. If you want to call */patient/2250f7ab-6517-4923-ac00-88ed26f85843* for a particular Bolt, you can use:

```
POST <internal-node-address>/internal/vcr/v2/search
{
  "query": {
    "@context": [
      "https://www.w3.org/2018/credentials/v1",
      "https://nuts.nl/credentials/v1"
    ]
  }
}
```

(continues on next page)

(continued from previous page)

```

    ],
    "type": ["VerifiableCredential", "NutsOrganizationCredential"],
    "credentialSubject": {
      "id": "did:nuts:JCJEi3waNGNhkmwVvFB3wdUsmDYPnTcZxYiWThZqgWKv",
      "purposeOfUse": "test-service",
      "resources": {
        "path": "/patient/2250f7ab-6517-4923-ac00-88ed26f85843"
      }
    }
  },
  "searchOptions": {
    "allowUntrustedIssuer": true
  }
}

```

This call will return all authorization credentials with a *purposeOfUse* equal to *test-service* and with which you are allowed to call the resource located at */patient/2250f7ab-6517-4923-ac00-88ed26f85843*. Any value in an authorization credential can be used as a param in the search API. The search *key* requires a valid JSON path expression.

### 6.3.1 Return values

When searching for authorization credentials, the credentials are returned as a verifiable credential. Most of the time, you'll only need the credential identifier, available in the root *id* field.

Example return value:

```

[
  {
    "@context": [
      "https://www.w3.org/2018/credentials/v1",
      "https://nuts.nl/credentials/v1"
    ],
    "credentialSubject": {
      "id": "did:nuts:JCJEi3waNGNhkmwVvFB3wdUsmDYPnTcZxYiWThZqgWKv",
      "legalBase": {
        "consentType": "implied"
      },
      "purposeOfUse": "test-service",
      "resources": [
        {
          "operations": [
            "read"
          ],
          "path": "/patient/2250f7ab-6517-4923-ac00-88ed26f85843",
          "userContext": true
        }
      ],
      "subject": "urn:oid:2.16.840.1.113883.2.4.6.3:123456780"
    },
    "id": "did:nuts:JCJEi3waNGNhkmwVvFB3wdUsmDYPnTcZxYiWThZqgWKv#314542e8-c8cc-4502-a7df-a815ac47c06b",
    "issuanceDate": "2021-07-26T14:36:10.163463+02:00",
  }
]

```

(continues on next page)

(continued from previous page)

```
    "issuer": "did:nuts:JCJEi3waNGNhkmwVvFB3wdUsmDYPnTcZxYiWThZqgWKv",
    "proof": {
      "created": "2021-07-26T14:36:10.163463+02:00",
      "jws": "eyJhbGciOiJFUzI1NiIsImI2NCI6ZmFsc2UsImNyaXQiOlsiYjY0Ii19..
↪k4cda7fMY05mnp4gsNJ3hNExjsSz3mqymyo4xJWkbb9-1URLjVWIzPg6R62T-YETV7UXvz1X9QteuhbmoM1JLA
↪",
      "proofPurpose": "assertionMethod",
      "type": "JsonWebSignature2020",
      "verificationMethod": "did:nuts:JCJEi3waNGNhkmwVvFB3wdUsmDYPnTcZxYiWThZqgWKv
↪#_3uOS5FqcyGj-cn-Yynv5epH0UVqbt_2BWXPfy0oKnU"
    },
    "type": [
      "NutsAuthorizationCredential",
      "VerifiableCredential"
    ]
  }
]
```



## GETTING STARTED ON EHR INTEGRATION

This getting started manual assumes the vendor and its clients (care organizations) are set up on the Nuts Network through *Getting Started on customer integration*. The next step is to integrate the vendor's electronic health record (EHR) with the Nuts node to execute Bolts.

All APIs used in the following chapters are documented at the [API](#) page. There you will also find the OpenAPI specifications for generating client code.

### 7.1 Resolving Bolt endpoints

Bolts define which technical endpoints should be defined for exchanging information. These endpoints are grouped as services which are generally named after the Bolt they support. The Nuts registry (as described by *Getting Started on customer integration*) can be queried to find care organizations that support a particular Bolt, and to resolve the technical endpoints associated with it.

#### 7.1.1 Searching organizations

To find care organizations (registered in the Nuts registry) that support a specific Bolt, the search organization API can be used. It takes a query parameter that's used to match organization names and optionally a service type (from its DID document). If the DID service type is supplied the API only returns organizations which DID Document has a service with that type.

For example, the following API call searches the Nuts registry for organizations which name matches “Ziekenhuis” and have a service of type “secure-direct-messaging” on their DID Document:

```
GET <internal-node-address>/internal/didman/v1/search/organizations?query=Ziekenhuis&
  didServiceType=secure-direct-messaging
```

---

**Note:** The example DID service type “secure-direct-messaging” could be defined by a (fictional) “Secure Direct Messaging” Bolt to be published by organizations that allow their employees to securely chat with other organizations through Nuts.

---

The API call returns a list of search results where each entry contains the organization and its last DID Document:

```
[
  {
    "didDocument": {
      "@context": "https://www.w3.org/ns/did/v1",
      "assertionMethod": [
```

(continues on next page)

(continued from previous page)

```

    "did:nuts:JCx4c3ufdKNgaZJ4h54AghY8ZgCznptNpjHUtzvVgcvW
    ↪#Cv0c4hlz4My7pKa6Wh6UN7gnTAXi5WUpNChqsUuIL1A"
  ],
  "controller": "did:nuts:5bSHwHtpSZfSCdCqaHvzDceEkjgNuKvTWVvQPB5DdeD9",
  "id": "did:nuts:JCx4c3ufdKNgaZJ4h54AghY8ZgCznptNpjHUtzvVgcvW",
  "verificationMethod": [
    /* etc */
  ],
  "service": {
    "type": "secure-direct-messaging",
    /* etc */
  }
},
"organization": {
  "city": "Doornenburg",
  "name": "Fort Pannerden"
}
}
]

```

For an organization to be returned as search results the following requirements must be met:

- It must have an active DID Document.
- The issuer of its verifiable credential (`NutsOrganizationCredential`) must be trusted by the local node.
- Its verifiable credential must not be expired or revoked.

The query parameter is used to phonetically match the organization name: it supports partial matches and matches that sound like the given query.

### 7.1.2 Resolving endpoints

When an organization has been selected, the next step is to resolve the technical endpoints. This is done by taking the compound service as specified by the Bolt and resolving its endpoint references to an actual URL endpoints. You can use the DIDMan `getCompoundServiceEndpoint` API operation for this.

## 7.2 Receiving Authorization Credentials

Some Bolts require authorization credentials to authenticate data exchanges. These credentials are distributed privately over an authenticated connection. To receive privately distributed credentials issued to your care organizations, the DID documents of the care organizations need to contain a `NutsComm` service that references the vendor's. See [setting up your node for a network](#) for how to achieve this.

## DECENTRALIZED IDENTIFIERS

Nuts uses [W3C Decentralized Identifiers](#) as a base for tracking identities. From the W3C website:

Decentralized identifiers (DIDs) are a new type of identifier that enables verifiable, decentralized digital identity. A DID identifies any subject (e.g., a person, organization, thing, data model, abstract entity, etc.) that the controller of the DID decides that it identifies. In contrast to typical, federated identifiers, DIDs have been designed so that they may be decoupled from centralized registries, identity providers, and certificate authorities. Specifically, while other parties might be used to help enable the discovery of information related to a DID, the design enables the controller of a DID to prove control over it without requiring permission from any other party. DIDs are URIs that associate a DID subject with a DID document allowing trustable interactions associated with that subject.

Within Nuts, DIDs identify both care organizations and software vendors. All DID methods require their own specification. The Nuts specification for the *nuts* DID method can be found on <https://nuts-specification.readthedocs.io>. Nuts DIDs are designed so they represent public/private key pairs. Any party can generate and claim a key pair. Only when information is added to the key pair, the key pair becomes important.

DIDs can gather claims through [Verifiable Credentials](#). This allows a DID to actually represent something known in real life. For example: adding an organization name credential connects the key pair to the name of the organization. It connects the digital world to the real world.

### 8.1 DID Documents

DIDs are backed by a *DID Document*. It defines the public keys, who can alter the document and any services related to the DID. DID documents are automatically propagated through the network when they are created. When DID documents are created, the DID **always** represents the public key fingerprint of the associated key. A DID document is always created with a new key, the holder of the key can delegate the control to another DID.

#### 8.1.1 Controller

The controller of the DID document is the only one that can change the contents. It can assign other controllers, change keys, change services and revoke the DID. When created, the DID document only has a single controller: the DID itself and the key related to it. It can choose to change add new controllers and remove existing ones. Changes to DID documents are only accepted when the network transaction is signed with a controller's **authentication** key.

### 8.1.2 Verification Method

All public keys within a Nuts DID Document are listed under **verificationMethod**.

### 8.1.3 Assertion Method

Keys referenced from the **assertionMethod** section are used to sign JWTs in the OAuth flow and for issuing *Verifiable Credentials*.

### 8.1.4 Authentication Method

Keys referenced from the **authentication** section are used to change the DID document and sign network transactions.

### 8.1.5 Services

The **services** section is used to list service endpoints. There are some endpoints that are shared amongst all services, like the **oauth** service. But most service endpoints will be coming from specific [Bolts](#).

## VERIFIABLE CREDENTIALS



## VC CONCEPT MAPPING

Verifiable Credentials are, by nature, very dynamic. This is reflected by the data format: either JSON or fields within a JWT (basically also JSON). There are two ways to cope with this: 1) transform every credential to a fixed concept. This creates a reliable DB but requires custom code for each credential type. Or 2) store the credential in a document store and provide a way to interact with the credentials through some interface where magic happens.

We opted for the latter and call it concept mappings. A *concept* represents a selection of data that is needed to support bolts. An example concept is a care organization. A key part of a care organization is its name. This name is given to a DID via a Verifiable Credential. Multiple credentials can give a DID its name, the concept mappings makes sure only a single interface is required at the side of the Nuts node.

See preconfigured concepts for a list of supported concepts.

### 10.1 Configuration

Credentials are configured to have indices and a template for transformation.

The organization credential configuration looks like this:

```
concept: organization
credentialType: NutsOrganizationCredential
public: true
indices:
  - name: index_id
    parts:
      - path: id
  - name: index_issuer
    parts:
      - path: issuer
  - name: index_subject
    parts:
      - path: credentialSubject.id
      alias: subject
  - name: index_name_city
    parts:
      - path: credentialSubject.organization.name
      alias: organization.name
      tokenizer: whitespace
      transformer: cologne
      - path: credentialSubject.organization.city
      alias: organization.city
```

(continues on next page)

(continued from previous page)

```
    tokenizer: whitespace
    transformer: cologne
template: |
  {
    "id": "<<id>>",
    "issuer": "<<issuer>>",
    "type": "NutsOrganizationCredential",
    "subject": "<<credentialSubject.id>>",
    "organization": {
      "name": "<<credentialSubject.organization.name>>",
      "city": "<<credentialSubject.organization.city>>"
    }
  }
}
```

It contains 4 main parts: *concept*, *credentialType*, *indices* and *template*. The `NutsOrganizationCredential` is mapped to the *organization* concept. When searching for credentials, you use the concept name. When creating a credential you use the credential type. When you resolve a credential, you'll always get the raw credential.

### 10.1.1 Public

By default credential types are regarded as private. This means they are only synchronized between the issuer and `credentialSubject`. It is possible to make credentials public by creating a template for the credential type and add the *public: true* setting. Credentials that are created without a template are regarded as private.

### 10.1.2 Indices

Each configuration may contain a list of indices. Each index has a name and contains a list of parts. The name of the index is used to identify the index. Each index is scoped to its own credential type. An index part must contain a *path*. The path is a JSON path query. The values found at that location will be used for indexing. See [go-leia](#) for the syntax and available options.

An index part may contain an *alias*. The alias is used as a search key. In the example above, the JSON path `credentialSubject.organization.name` has an alias of `organization.name`. This means that in a search query the key `organization.name` can be used. Aliases are not unique to a credential type. This allows for searching over multiple credential types with a single query.

An index part can also contain a transformer and/or a tokenizer. A tokenizer will split a value into multiple values to be indexed. For example: the `whitespace` tokenizer will split a sentence into a list of words to be indexed. The name *healthcare organization the Nutty professor* will be split and can be found by just searching for *healthcare*, *organization*, *the*, *Nutty* or *professor*. Possible values for `tokenizer`: *whitespace*.

A transformer transforms a value before the value is used as an index key. A transformer will also transform search parameters that use that index. For example: the `lowercase` transformer will transform all indexed values and search params to lowercase. The word *Nutty* in the previous example could then be found with *nutty*, *NUTTY* or any other combination of upper- and lowercase letters. Possible values for `transformer`: *lowercase*, *cologne*.

The *cologne* transformer is a phonetic transformer. It's like soundex but works better for germanic type languages.



### 10.1.3 Template

The template is used to transform a credential to a common output format when searching for a concept. Values that start with << and end with >> are JSON path expressions. They are replaced with the result of the JSON path expression when applied to found credentials. A template is optional. When not defined, the raw credential is returned.

Using arrays is currently not supported in a template.



## NUTS NODE DEVELOPMENT

### 11.1 Requirements

Go  $\geq 1.17$  is required.

### 11.2 Building

Just use `go build`.

### 11.3 Running tests

Tests can be run by executing

```
go test ./...
```

### 11.4 Code Generation

Code generation is used for generating mocks, OpenAPI client- and servers, and gRPC services. Make sure that `GOPATH/bin` is available on `PATH` and that the dependencies are installed

Install `protoc`:

MacOS: `brew install protobuf`  
Linux: `apt install -y protobuf-compiler`

Install Go tools:

```
make install-tools
```

Generating code:

To regenerate all code run the `run-generators` target from the makefile or use one of the following for a specific group

Group	Command
Mocks	<code>make gen-mocks</code>
OpenApi	<code>make gen-api</code>
Protobuf + gRPC	<code>make gen-protobuf</code>
All	<code>make run-generators</code>

## 11.5 Docs Generation

To generate the documentation, you'll need `python3`, `sphinx` and a bunch of other stuff. After you have installed `python3` (and `pip3` if this not already installed) run

```
pip3 install -r docs/requirements.txt
```

### 11.5.1 README

The readme is auto-generated from a template and uses the documentation to fill in the blanks.

```
make gen-readme
```

### 11.5.2 Documentation

The documentation can be build by running the following command from the `/docs` directory:

```
make html
```

## API DEVELOPMENT

When developing APIs, please follow these guidelines.

### 12.1 Contract first

The Nuts node APIs are specified in **Open API Specification (OAS)**<https://swagger.io/specification/>. The files are located under `/docs/_static/<engine>/<version>.yaml`. Where `<engine>` is a specific module like `crypto` or `auth` and `<version>` defines the version of the API. We use version `3.0.y` of the OAS.

#### 12.1.1 Versioning

We use versioning of the APIs. This is reflected in both the OAS files and the HTTP paths. Versions must follow the pattern `v` and start at `v1`. These are major versions, any breaking change results in a new major version of the API. New additions, bug fixes and changes that are backwards compatible may be done in the current version.

#### 12.1.2 Code generation

The OAS files are used for code generation. The makefile contains the `gen-api` target which will generate the code. The build target only needs to be extended when a new version or new engine is added. Generated code is always placed in `/<engine>/api/<version>/generated.go`.

#### 12.1.3 Return codes

The error return values are generalized for all API calls. The return values follow [RFC7807](#). The definition is available under `/docs/_static/common/error_response.yaml`. The error definition can be used in a OAS file:

```
paths:
  /some/path:
    get:
      responses:
        default:
          $ref: '../common/error_response.yaml'
```

The error responses will not be listed as responses in the online generated documentation. To describe error responses, the specific responses need to be added to the API description:

```
paths:
  /some/path:
    post:
      description: |
        Some description on the API

      error returns:
        * 400 - incorrect input
```

## 12.2 Paths

The API paths are designed so different security schemes can be setup easily.

API paths follow the following pattern:

```
/<context>/<engine>/<version>/<action>
```

All paths start with a security <context>:

- `/internal/**` These APIs are meant to be behind a firewall and should only be available to the internal infrastructure. All DID Document manipulation APIs fall under this category.
- `/n2n/**` These APIs must be available to other nodes from the network. This means they must be protected with the required client certificate as specified by [RFC011](#). The creation of an access token is one example of such an API.
- `/public/**` These APIs must be publicly available on a valid domain. No security must be set. These APIs are used by mobile devices.

After the context, the <engine> is expected. An engine defines a logical unit of functionality. Each engine has its own OAS file. Then as discussed earlier, the <version> is expected. The last part is the <action>, this part can be freely chosen in a RESTful manor.

## ISSUING AND SEARCHING VERIFIABLE CREDENTIALS

### 13.1 Issuing VCs

As a node, you can issue credentials with each DID you control (whether they are trusted is a different story). A credential is issued through the API or CLI. The node will add sensible defaults for:

- @context
- id
- issuanceDate
- proof

You are required to provide the *credentialSubject*, the *issuer*, the *type* and an optional *expirationDate*. So calling `/internal/vcr/v2/issuer/vc` with

```
{
  "issuer": "did:nuts:ByJvBu2Ex21tNdn5s8FBnqmRBTCGkqRHms5ci7gKM8rg",
  "type": "NutsOrganizationCredential",
  "credentialSubject": {
    "id": "did:nuts:9UKf9F9sRtiq4gR3bxfGQAeARtJeU8jvPqfWJcFP6ziN",
    "organization": {
      "name": "Because we care B.V.",
      "city": "IJbergen"
    }
  },
  "visibility": "public"
}
```

Will be expanded by the node to:

```
{
  "context": [
    "https://www.w3.org/2018/credentials/v1",
    "https://nuts.nl/credentials/v1"
  ],
  "credentialSubject": {
    "id": "did:nuts:9UKf9F9sRtiq4gR3bxfGQAeARtJeU8jvPqfWJcFP6ziN",
    "organization": {
      "city": "IJbergen",
      "name": "Because we care B.V."
    }
  }
}
```

(continues on next page)

(continued from previous page)

```

    },
    "id": "did:nuts:ByJvBu2Ex21tNdn5s8FBnqmRBTCGkqRHms5ci7gKM8rg#a1d8ee3f-f404-44d5-bd07-
↪71d3b144ce54",
    "issuanceDate": "2021-03-05T09:37:05.732811+01:00",
    "issuer": "did:nuts:ByJvBu2Ex21tNdn5s8FBnqmRBTCGkqRHms5ci7gKM8rg",
    "proof": {
      "created": "2021-03-05T09:37:05.732811+01:00",
      "jws": "eyJhbGciOiJFUzI1NiIsImI2NCI6ZmFsc2UsImNyaXQiOlsiYjY0Il19..
↪s6lxJa7pOpqlhcWhJoKRMJIJiD4i+IUkfmhy+rUvNzZayVHAq+lZaFxBsv9rQCe0ewpZq/
↪6z3hSUOURo6mnHhg==",
      "proofPurpose": "assertionMethod",
      "type": "JsonWebSignature2020",
      "verificationMethod": "did:nuts:ByJvBu2Ex21tNdn5s8FBnqmRBTCGkqRHms5ci7gKM8rg
↪#gSEtbS2dOsS9PSrV13RwaZH3Ps60TI14GvLx8dPqgQ"
    },
    "type": [
      "NutsOrganizationCredential",
      "VerifiableCredential"
    ]
  }
}

```

The *visibility* property indicates the contents of the VC are published on the network, so it can be read by everyone.

## 13.2 Searching VCs

You can search for VCs by providing a VC which should be used for matching in JSON-LD format. Searching works by posting a Verifiable Credential to `/internal/vcr/v2/search` that contains fields to match. The operation yields an array containing the matched verifiable credentials.

The example below searches for a *NutsOrganizationCredential* (note that the *query* field contains the credential):

```

{
  "query": {
    "@context": [
      "https://www.w3.org/2018/credentials/v1",
      "https://nuts.nl/credentials/v1"
    ],
    "type": ["VerifiableCredential", "NutsOrganizationCredential"],
    "credentialSubject": {
      "id": "did:nuts:SKUYi2g88ohjhiu49Q13ZWXvp678sjNiM7UHUCMyw",
      "organization": {
        "name": "Because we care B.V.",
        "city": "IJbergen"
      }
    }
  }
}

```

Note the fields *@context* and *type*, these are required for making it a valid VC in JSON-LD. In the example above they also contain Nuts specific contexts and types (since we're searching for a Nuts VC).

By default only VCs from trusted issuers are returned. You can specify the *searchOptions* field to include VCs from



untrusted issuers.



## EVENTS

Each event consists of a single JSON encoded message that is categorized by its subject which in turn are grouped into streams. Each stream defines how to handle limits, storage, data retention, deliverability etc.

### 14.1 Streams

Name	Summary	Policy	Durable	Message limit	Storage
nuts-disposable	Main event-stream	When the stream is full old messages will be discarded	No	100	Memory



## NUTS NODE APIS

Below you can discover the Nuts Node APIs and download their OpenAPI specifications:

- [Common](#) (required by others)
- [DID Manager](#)
- [Crypto](#)
- [Verifiable Credential Registry \(v1, deprecated: please use v2\)](#)
- [Verifiable Credential Registry \(v2\)](#)
- [Verifiable Data Registry](#)
- [Network](#)
- [Auth](#)



## NUTS NODE CONFIG

The Nuts node can be configured using a YAML configuration file, environment variables and commandline params.

The parameters follow the following convention: `$ nuts --parameter X` is equal to `$ NUTS_PARAMETER=X nuts` is equal to `parameter: X` in a yaml file.

Or for this piece of yaml

```
nested:
  parameter: X
```

is equal to `$ nuts --nested.parameter X` is equal to `$ NUTS_NESTED_PARAMETER=X nuts`

Config parameters for engines are prepended by the `engine.ConfigKey` by default (configurable):

```
engine:
  nested:
    parameter: X
```

is equal to `$ nuts --engine.nested.parameter X` is equal to `$ NUTS_ENGINE_NESTED_PARAMETER=X nuts`

While most options are a single value, some are represented as a list (indicated with the square brackets in the table below). To provide multiple values through flags or environment variables you can separate them with a comma (,).

### 16.1 Ordering

Command line parameters have the highest priority, then environment variables, then parameters from the configfile and lastly defaults. The location of the configfile is determined by the environment variable `NUTS_CONFIGFILE` or the commandline parameter `--configfile`. If both are missing the default location `./nuts.yaml` is used.

### 16.2 Server options

The following options can be configured on the server:

Key	Default	Description
configfile	nuts.yaml	Nuts config file
datadir	./data	Directory where the node stores its files.
loggerformat	text	Log format (text, json)
strictmode	false	When set, insecure settings are forbidden.
verbosity	info	Log level (trace, debug, info, warn, error)

Table 1 –

Key	Default	Description
http.default.address	:1323	Address and port the server will be listening to
http.default.cors.origin	[]	When set, enables CORS from the specified origins for the on default
<b>Auth</b>		
auth.clockskew	5000	Allowed JWT Clock skew in milliseconds
auth.contractvalidators	[irma,uzi,dummy]	sets the different contract validators to use
auth.http.timeout	30	HTTP timeout (in seconds) used by the Auth API HTTP client
auth.irma.autoupdateschemas	true	set if you want automatically update the IRMA schemas every 60 m
auth.irma.schememanager	pbf	IRMA schemeManager to use for attributes. Can be either 'pbf' or
auth.publicurl		public URL which can be reached by a users IRMA client, this sho
<b>Crypto</b>		
crypto.storage	fs	Storage to use, 'fs' for file system, vaultkv for Vault KV store, defa
crypto.vault.address		The Vault address. If set it overwrites the VAULT_ADDR env var.
crypto.vault.pathprefix	kv	The Vault path prefix. default: kv.
crypto.vault.token		The Vault token. If set it overwrites the VAULT_TOKEN env var.
<b>Event manager</b>		
events.nats.hostname	localhost	Hostname for the NATS server
events.nats.port	4222	Port where the NATS server listens on
events.nats.storagedir		Directory where file-backed streams are stored in the NATS server
events.nats.timeout	30	Timeout for NATS server operations
<b>Network</b>		
network.bootstrapnodes	[]	List of bootstrap nodes (<host>:<port>) which the node initially c
network.certfile		PEM file containing the server certificate for the gRPC server. Requ
network.certkeyfile		PEM file containing the private key of the server certificate. Requir
network.disablenodeauthentication	false	Disable node DID authentication using client certificate, causing al
network.enablediscovery	true	Whether to enable automatic connecting to other nodes.
network.enabletls	true	Whether to enable TLS for incoming and outgoing gRPC connectio
network.grpcaddr	:5555	Local address for gRPC to listen on. If empty the gRPC server wor
network.nodedid		Specifies the DID of the organization that operates this node, typica
network.truststorefile		PEM file containing the trusted CA certificates for authenticating re
network.v1.advertiagnosticsinterval	5000	Interval (in milliseconds) that specifies how often the node should l
network.v1.adverthashesinterval	2000	Interval (in milliseconds) that specifies how often the node should l
network.v1.collectmissingpayloadsinterval	60000	Interval (in milliseconds) that specifies how often the node should c
network.v2.gossipinterval	5000	Interval (in milliseconds) that specifies how often the node should g
<b>VCR</b>		
vcr.overrideissueallpublic	true	Overrides the "Public" property of a credential when issuing creden

This table is automatically generated using the configuration flags in the core and engines. When they're changed the options table must be regenerated using the Makefile:

```
$ make update-docs
```



## CONTRIBUTE

If you want to contribute to any of the nuts foundation projects or to this documentation, please fork the correct project from [Github](#) and create a pull-request.

### 17.1 Documentation contributions

Documentation is written in Restructured Text. A CheatSheet can be found [here](#).

You can test your documentation by installing the required components.

### 17.2 Documentation initialisation

When starting a new project, the documentation can be initialised using:

```
sphinx-quickstart docs
```

This will start the interactive setup of sphinx with a document root at *docs*. For Nuts projects we use that specific directory for documentation in a code project. You might have noticed that the *nuts-documentation* repo uses the root directory as documentation root.

Most defaults will do, although we use intersphinx to go back-and-forth between the different sub-projects.



## RELEASE NOTES

Whats has been changed, and how to update between versions.

### 18.1 Almond (v1.0.0)

Release date: 2022-04-01

This is the initial release of the Nuts node reference implementation. It implements RFC001 - RFC016 specified by the [Nuts specification](#). This release is intended for developers. It contains a stable API that will be backwards compatible for the next versions. The releases until the first production release will mainly focus on network and Ops related features.

To start using this release, please consult the getting started section.

#### 18.1.1 Features / improvements

Future releases will list new features and improvements that have been added since the previous release.

#### 18.1.2 Dropped features

New major releases might drop support for features that have been deprecated in a previous release. Keep an eye on this section for every release.

#### 18.1.3 Deprecated features

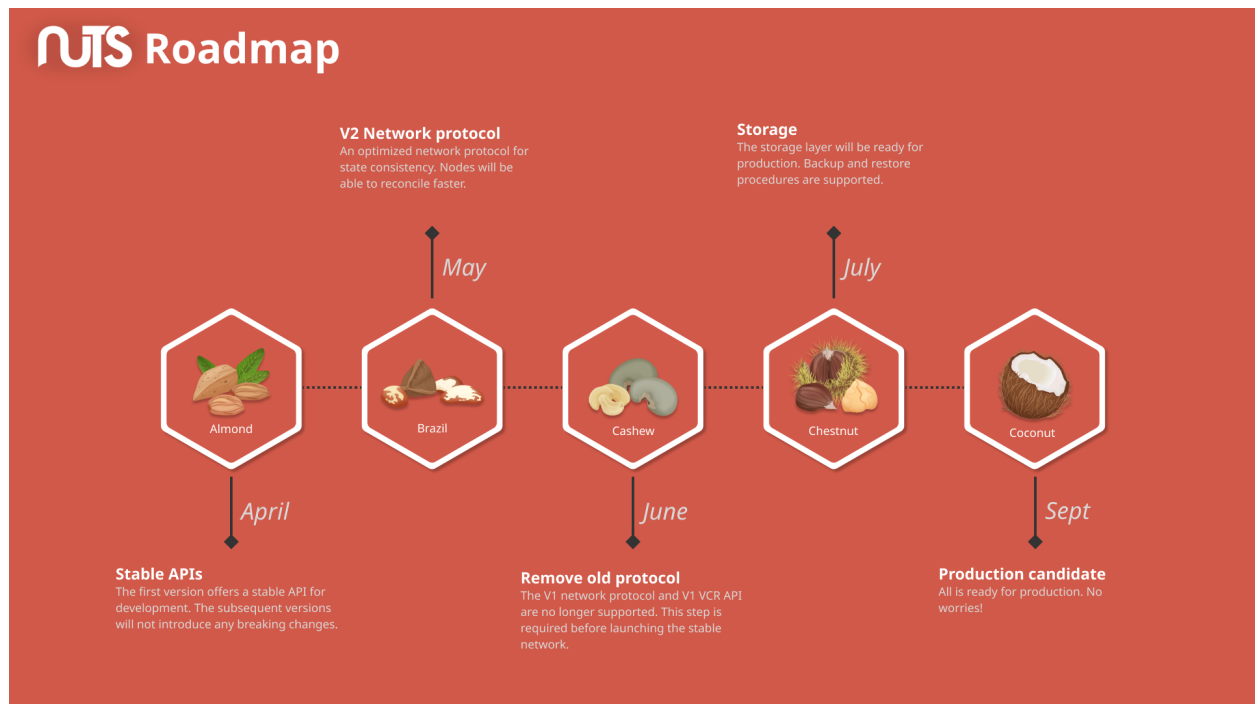
Some features will be deprecated because they have been succeeded by an improved version or when they are no longer used. Removing old code helps in reducing maintenance costs of the code base. Features that are marked as *deprecated* will be listed here. Any vendor using these features will have until next version to migrate to the alternative. Keep an eye on this section for every release.

- VCR V1 API is deprecated and will be removed in the next release. Please migrate all calls to the V2 API.

### 18.1.4 Bugfixes

This section contains a list of bugfixes. It'll match resolved Github issues with the **bug** tag.

ROADMAP





## NUTS NODE MONITORING

### 20.1 Basic service health

A status endpoint is provided to check if the service is running and if the web server has been started. The endpoint is available over http so it can be used by a wide range of health checking services. It does not provide any information on the individual engines running as part of the executable. The main goal of the service is to give a YES/NO answer for if the service is running?

```
GET /status
```

It'll return an "OK" response and a 200 status code.

### 20.2 Basic diagnostics

```
GET /status/diagnostics
```

It'll return some text displaying the current status of the various services:

```
Status
  Registered engines: [Status Logging]
Logging
  verbosity: INFO
```

If you supply *application/json* for the *Accept* HTTP header it will return the diagnostics in JSON format.

### 20.3 Metrics

The Nuts service executable has build-in support for **Prometheus**. Prometheus is a time-series database which supports a wide variety of services. It also allows for exporting metrics to different visualization solutions like **Grafana**. See <https://prometheus.io/> for more information on how to run Prometheus. The metrics are exposed at `/metrics`

### 20.3.1 Configuration

In order for metrics to be gathered by Prometheus. A job has to be added to the `prometheus.yml` configuration file. Below is a minimal configuration file that will only gather Nuts metrics:

```
# my global config
global:
  scrape_interval:     15s # Set the scrape interval to every 15 seconds. Default is
    ↪ every 1 minute.
  evaluation_interval: 15s # Evaluate rules every 15 seconds. The default is every 1
    ↪ minute.
  # scrape_timeout is set to the global default (10s).

# Load rules once and periodically evaluate them according to the global 'evaluation_
    ↪ interval'.
rule_files:
# - "first_rules.yml"
# - "second_rules.yml"

# A scrape configuration containing exactly one endpoint to scrape:
# Here it's Prometheus itself.
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeseries scraped from this
    ↪ config.
  - job_name: 'nuts'
    metrics_path: '/metrics'
    scrape_interval: 5s
    static_configs:
      - targets: ['127.0.0.1:1323']
```

It's imported to enter the correct IP/domain and port where the Nuts node can be found!

### 20.3.2 Exported metrics

The Nuts service executable exports the following metrics by default. These cover the basic needs for monitoring the process and http layer.

```
# HELP go_gc_duration_seconds A summary of the pause duration of garbage collection.
    ↪ cycles.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 4.08e-05
go_gc_duration_seconds{quantile="0.25"} 6.25e-05
go_gc_duration_seconds{quantile="0.5"} 8.44e-05
go_gc_duration_seconds{quantile="0.75"} 0.0001046
go_gc_duration_seconds{quantile="1"} 0.0004961
go_gc_duration_seconds_sum 0.0016542
go_gc_duration_seconds_count 14
# HELP go_goroutines Number of goroutines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 79
# HELP go_info Information about the Go environment.
# TYPE go_info gauge
go_info{version="go1.13.12"} 1
```

(continues on next page)



(continued from previous page)

```

# HELP go_memstats_alloc_bytes Number of bytes allocated and still in use.
# TYPE go_memstats_alloc_bytes gauge
go_memstats_alloc_bytes 9.284216e+06
# HELP go_memstats_alloc_bytes_total Total number of bytes allocated, even if freed.
# TYPE go_memstats_alloc_bytes_total counter
go_memstats_alloc_bytes_total 6.929336e+07
# HELP go_memstats_buck_hash_sys_bytes Number of bytes used by the profiling bucket hash.
# TYPE go_memstats_buck_hash_sys_bytes gauge
go_memstats_buck_hash_sys_bytes 1.477216e+06
# HELP go_memstats_frees_total Total number of frees.
# TYPE go_memstats_frees_total counter
go_memstats_frees_total 394819
# HELP go_memstats_gc_cpu_fraction The fraction of this program's available CPU time
# TYPE go_memstats_gc_cpu_fraction gauge
go_memstats_gc_cpu_fraction 0.0005164729882960839
# HELP go_memstats_gc_sys_bytes Number of bytes used for garbage collection system
# TYPE go_memstats_gc_sys_bytes gauge
go_memstats_gc_sys_bytes 2.394112e+06
# HELP go_memstats_heap_alloc_bytes Number of heap bytes allocated and still in use.
# TYPE go_memstats_heap_alloc_bytes gauge
go_memstats_heap_alloc_bytes 9.284216e+06
# HELP go_memstats_heap_idle_bytes Number of heap bytes waiting to be used.
# TYPE go_memstats_heap_idle_bytes gauge
go_memstats_heap_idle_bytes 5.24288e+07
# HELP go_memstats_heap_inuse_bytes Number of heap bytes that are in use.
# TYPE go_memstats_heap_inuse_bytes gauge
go_memstats_heap_inuse_bytes 1.2255232e+07
# HELP go_memstats_heap_objects Number of allocated objects.
# TYPE go_memstats_heap_objects gauge
go_memstats_heap_objects 32515
# HELP go_memstats_heap_released_bytes Number of heap bytes released to OS.
# TYPE go_memstats_heap_released_bytes gauge
go_memstats_heap_released_bytes 4.8848896e+07
# HELP go_memstats_heap_sys_bytes Number of heap bytes obtained from system.
# TYPE go_memstats_heap_sys_bytes gauge
go_memstats_heap_sys_bytes 6.4684032e+07
# HELP go_memstats_last_gc_time_seconds Number of seconds since 1970 of last garbage
# TYPE go_memstats_last_gc_time_seconds gauge
go_memstats_last_gc_time_seconds 1.5942182098267434e+09
# HELP go_memstats_lookups_total Total number of pointer lookups.
# TYPE go_memstats_lookups_total counter
go_memstats_lookups_total 0
# HELP go_memstats_mallocs_total Total number of mallocs.
# TYPE go_memstats_mallocs_total counter
go_memstats_mallocs_total 427334
# HELP go_memstats_mcache_inuse_bytes Number of bytes in use by mcache structures.
# TYPE go_memstats_mcache_inuse_bytes gauge
go_memstats_mcache_inuse_bytes 13888

```

(continues on next page)

(continued from previous page)

```

# HELP go_memstats_mcache_sys_bytes Number of bytes used for mcache structures obtained
↳from system.
# TYPE go_memstats_mcache_sys_bytes gauge
go_memstats_mcache_sys_bytes 16384
# HELP go_memstats_mspan_inuse_bytes Number of bytes in use by mspan structures.
# TYPE go_memstats_mspan_inuse_bytes gauge
go_memstats_mspan_inuse_bytes 115736
# HELP go_memstats_mspan_sys_bytes Number of bytes used for mspan structures obtained
↳from system.
# TYPE go_memstats_mspan_sys_bytes gauge
go_memstats_mspan_sys_bytes 229376
# HELP go_memstats_next_gc_bytes Number of heap bytes when next garbage collection will
↳take place.
# TYPE go_memstats_next_gc_bytes gauge
go_memstats_next_gc_bytes 1.6785728e+07
# HELP go_memstats_other_sys_bytes Number of bytes used for other system allocations.
# TYPE go_memstats_other_sys_bytes gauge
go_memstats_other_sys_bytes 1.584792e+06
# HELP go_memstats_stack_inuse_bytes Number of bytes in use by the stack allocator.
# TYPE go_memstats_stack_inuse_bytes gauge
go_memstats_stack_inuse_bytes 2.424832e+06
# HELP go_memstats_stack_sys_bytes Number of bytes obtained from system for stack
↳allocator.
# TYPE go_memstats_stack_sys_bytes gauge
go_memstats_stack_sys_bytes 2.424832e+06
# HELP go_memstats_sys_bytes Number of bytes obtained from system.
# TYPE go_memstats_sys_bytes gauge
go_memstats_sys_bytes 7.2810744e+07
# HELP go_threads Number of OS threads created.
# TYPE go_threads gauge
go_threads 18
# HELP process_cpu_seconds_total Total user and system CPU time spent in seconds.
# TYPE process_cpu_seconds_total counter
process_cpu_seconds_total 2.58
# HELP process_max_fds Maximum number of open file descriptors.
# TYPE process_max_fds gauge
process_max_fds 1.048576e+06
# HELP process_open_fds Number of open file descriptors.
# TYPE process_open_fds gauge
process_open_fds 25
# HELP process_resident_memory_bytes Resident memory size in bytes.
# TYPE process_resident_memory_bytes gauge
process_resident_memory_bytes 4.5256704e+07
# HELP process_start_time_seconds Start time of the process since unix epoch in seconds.
# TYPE process_start_time_seconds gauge
process_start_time_seconds 1.59421820085e+09
# HELP process_virtual_memory_bytes Virtual memory size in bytes.
# TYPE process_virtual_memory_bytes gauge
process_virtual_memory_bytes 1.37965568e+08
# HELP process_virtual_memory_max_bytes Maximum amount of virtual memory available in
↳bytes.
# TYPE process_virtual_memory_max_bytes gauge

```

(continues on next page)

(continued from previous page)

```
process_virtual_memory_max_bytes -1
# HELP promhttp_metric_handler_requests_in_flight Current number of scrapes being served.
# TYPE promhttp_metric_handler_requests_in_flight gauge
promhttp_metric_handler_requests_in_flight 1
# HELP promhttp_metric_handler_requests_total Total number of scrapes by HTTP status_
↪code.
# TYPE promhttp_metric_handler_requests_total counter
promhttp_metric_handler_requests_total{code="200"} 0
promhttp_metric_handler_requests_total{code="500"} 0
promhttp_metric_handler_requests_total{code="503"} 0
```

### 20.3.3 Network DAG Visualization

All network transactions form a directed acyclic graph (DAG) which helps achieving consistency and data completeness. Since it's a hard to debug, complex structure, the network API provides a visualization which can be queried from */internal/network/v1/diagnostics/graph*. It is returned in the *dot* format which can then be rendered to an image using *dot* or *graphviz* (given you saved the output to *input.dot*):

```
dot -T png -o output.png input.dot
```



## ADMINSTERING YOUR NODE

The Nuts executable this project provides can be used to both run a Nuts server (a.k.a. node) and administer a running node remotely. This chapter explains how to administer your running Nuts node.

### 21.1 Prerequisites

The following is needed to run a Nuts node:

1. Nuts executable for your platform.
2. The address of your running Nuts node. You can pass this using the *address* variable.

### 21.2 Commands

Run the executable without command or flags, or with the *help* command to find out what commands are supported:

```
$ nuts
```

For example, to list all network transactions in your node (replace the value of *NUTS\_ADDRESS* with the HTTP address of your Nuts node):

```
$ NUTS_ADDRESS=my-node:1323 nuts network list
```

The following options can be supplied when running CLI commands:

Key	Default	Description
ad- dress	local- host:1323	Address of the remote node. Must contain at least host and port, URL scheme may be omitted. In that case it ' <a href="#">http://</a> ' is prepended.
time- out	10s	Client time-out when performing remote operations, such as '500ms' or '10s'. Refer to Golang's 'time.Duration' syntax for a more elaborate description of the syntax.



## CONFIGURING FOR PRODUCTION

Running a Nuts node in a production environment has additional requirements regarding security and data integrity compared to development or test environments. This page instructs how to *configure* your node for running in a production environment and what to consider.

### 22.1 Persistence

All data the node produces is stored on disk in the configured data directory (*datadir*). It is recommended to backup everything in that directory.

The private keys are stored in a storage backend. Currently 2 options are available.

#### 22.1.1 Vault

This storage backend is the recommended way of storing secrets. It uses the [Vault KV version 1 store](#). The prefix defaults to *kv* and can be configured using the *crypto.vault.pathprefix* option. There needs to be a KV Secrets Engine (v1) enabled under this prefix path.

All private keys are stored under the path *<prefix>/nuts-private-keys/\**. Each key is stored under the kid, resulting in a full key path like *kv/nuts-private-keys/did:nuts:123#abc*. A Vault token must be provided by either configuring it using the config *crypto.vault.token* or setting the *VAULT\_TOKEN* environment variable. The token must have a vault policy which enables *READ* and *WRITES* rights on the path. In addition it needs to *READ* the token information “auth/token/lookup-self” which should be part of the default policy.

#### 22.1.2 Filesystem

This is the default backend but not recommended for production. It stores keys unencrypted on disk. Make sure to include the directory in your backups and keep these on a safe place. If you want to use filesystem in strict-mode, you have to set it explicitly, otherwise the node fails during startup.

## 22.2 Strict mode

By default the node runs in a mode which allows the operator run configure the node in such a way that it is less secure. For production it is recommended to enable *strictmode* which blocks some of the unsafe configuration options (e.g. using the IRMA demo scheme).

## 22.3 HTTP Interface Binding

By default all HTTP endpoints get bound on `:1323` which generally isn't usable for production, since some endpoints are required to be accessible by the public and others only meant for administrator or your own XIS. You can determine the intended public by looking at the first part of the URL.

- Endpoints that start with `/public` should be accessible by the general public,
- `/internal` is meant for XIS application integration and administrators.

It's advisable to make sure internal endpoints aren't reachable from public networks. The HTTP configuration facilitates this by allowing you to bind sets of endpoints to a different HTTP port. This is done through the *http* configuration:

```
http:
# The following is the default binding which endpoints are bound to,
# which don't have an alternative bind specified under `alt`. Since it's a default it
↪ can be left out or
# be used to override the default bind address.
default:
  address: :1323
alt:
# The following binds all endpoints starting with `/internal` to `internal.lan:1111`
internal:
  address: internal.lan:1111
# The following binds all endpoints starting with `/public` to `nuts.vendor.nl:443`
public:
  address: nuts.vendor.nl:443
  # The following enables cross-domain requests (CORS) from irma.vendor.nl
  cors:
    origin:
      - irma.vendor.nl
# The following binds all endpoints starting with `/status` to all interfaces on `:80`
status:
  address: :80
```

### 22.3.1 Cross Origin Resource Sharing (CORS)

In some deployments CORS can be required for the public IRMA authentication endpoints when the user-facing authentication page is hosted on a (sub)domain that differs from Nuts Node's IRMA backend. CORS can be enabled on a specific HTTP interface by specifying the domains allowed to make CORS requests as *cors.origin* (see the example above). Although you can enable CORS on the default endpoint it's not advised to do so in a production environment, because CORS itself opens up new attack vectors on node administrators.



## 22.4 Diagnostics

To aid problem diagnosis every node in a network should share some information about itself; the type and version of software it's running, which peers it is connected to and how long it's been up. This helps others diagnosing issues when others experience communication problems with your, and other nodes. Although discouraged, this can be disabled by specifying *0* for *network.advertiagnosticsinterval*.

## 22.5 Nuts Network SSL/TLS Deployment Layouts

This section describes which deployment layouts are supported regarding SSL/TLS. In all layouts there should be X.509 server and client certificates issued by a Certificate Authority trusted by the network, if the node operator wants other Nuts nodes to be able to connect to the node and vice versa.

This is the simplest layout where the Nuts node is directly accessible from the internet:

This layout has the following requirements:

- X.509 server certificate and private key must be configured on the Nuts node.
- SSL/TLS terminator must use the trust anchors as specified by the network as root CA trust bundle.

In this layout incoming TLS traffic is decrypted on a SSL/TLS terminator and then being forwarded to the Nuts node. This layout is typically used to provide layer 7 load balancing and/or securing traffic “at the gates”:

This layout has the following requirements:

- X.509 server certificate and private key must be present on the SSL/TLS terminator.
- X.509 client certificate must be configured on the Nuts node.
- SSL/TLS terminator must use the trust anchors as specified by the network as root CA trust bundle.

In this layout incoming TLS traffic is forwarded to the Nuts node without being decrypted:

Requirements are the same as for the Direct WAN Connection layout.



## CONTACT

### 23.1 Information

More information about the Nuts foundation can be found at [nuts.nl](https://nuts.nl)

### 23.2 Communication

The main means of communication is via [Slack](#).