Nuts Documentation

Nuts community

Oct 02, 2020

Technology:

CHAPTER 1

Architecture

1.1 Certificate structure

Although Nuts promotes a decentralized architecture and uses individual signatures as proofs, it's still quite hard to secure the network without using two-way TLS. Corda, for example, requires a certificate tree structure for its permissioned network. Also vendors should be enabled to block other vendors if they think those vendors are compromised. They are responsible for the security, so they should be able to act on it. So the need for a root CA can be derived from these requirements. Access to the key will be restricted to the Nuts foundation. This does not give the foundation super powers since vendors and users (both validated by Irma signatures) are still part of the security scheme to access data.

If certificates are issued for any other environment than production, it is indicated in the common name. If the certificate is issued for the production environment, the environment part is omitted.

1.1.1 Vendor CA Certificate

The Vendor CA Certificate is the vendor's root certificate which is only used to issue its intermediate CA certificate. The key should be kept offline.

The certificate contains the 'Nuts domain' extension, indicating domain the vendor operates in. It also contains a subject alternative name containing the vendor ID (1.3.6.1.4.1.54851.4) corresponding with the vendor's ID in the registry.

1.1.2 Vendor CA Intermediate Certificate

The Vendor CA Intermediate certificate is used by the vendor on a day-to-day basis to issue certificates.

Note: The Nuts domain extension indicates what kind of node it is. It can be medical, social, insurance or private. Some of these domains are not allowed to process Social Security Numbers (BSN).

Extension	Critical?	Value
Subject		CN=<Vendor> CA <Environment>,O=<Vendor>,C=NL
BasicConstraints	Yes	CA=true pathLenConstraint=1
KeyUsage	Yes	digitalSignature & keyCertSign & crlSign
SubjectAltName	No	Vendor identifier, otherName: 1.3.6.1.4.1.54851.4=UTF8String.<number>
CRLDistributionPoints	No	Distribution point (URL) of the CRL
Nuts domain (1.3.6.1.4.1.54851.3)	No	UTF8String=healthcare | social | pgo | insurance

Vendor Certificate

This certificate is issued by the vendor to itself to sign events. The key pair and certificate are ephemeral, meaning they're valid just long enough to span the moment of signing. In any case the validity period should not be longer

than 1 minute (60 seconds). The advantage of this is that the CA doesn't need to care about maintaining a CRL for these certificates, since if the private key leaks, the validity period of the corresponding certificate is too short to be useful for exploits.

Extension	Critical?	Value
Subject		CN=<Vendor> <Environment>,O=<Vendor>,C=NL
KeyUsage	Yes	digitalSignature
SubjectAltName	No	Vendor identifier, otherName: 1.3.6.1.4.1.54851.4=UTF8String.<number>
CRLDistributionPoints	No	Distribution point (URL) of the CRL

Organisation Certificates

The organisation certificate is issued by the vendor when it claims a care organisation as their client. It is used to sign/encrypt JWTs and as KEK for encrypted data exchange with other vendors.

Extension	Critical?	Value
Subject		CN=<Organisation>,O=<Vendor>,C=NL
KeyUsage	Yes	digitalSignature, keyEncipherment, dataEncipherment
SubjectAltName	No	Organisation identifier, otherName: 2.16.840.1.113883.2.4.6.1=UTF8String.<number>
CRLDistributionPoints	No	Distribution point (URL) of the CRL
Nuts domain (1.3.6.1.4.1.54851.3)	No	UTF8String=healthcare | social | pgo | insurance

TLS Certificates

TLS certificates are issued by the vendor itself and are used to validate incoming requests. The implementation details are therefore a recommendation, since non-compliance imposes a risk limited to the vendor's software, rather than the whole Nuts network. Adhering to these recommendations also helps to establish a common denominator for implementations across the network.

Vendor TLS CA Certificate

This is a intermediate CA certificate, used to issue TLS client certificates. The certificate is issued by the Vendor CA.

Extension / Field	Critical?	Value
Subject		CN=<Vendor> TLS CA <Environment>,O=<Vendor>,C=NL
BasicConstraints	Yes	CA=true pathLenConstraint=0
KeyUsage	Yes	digitalSignature, keyCertSign, crlSign
SubjectAltName	No	Vendor identifier, otherName: 1.3.6.1.4.1.54851.4=UTF8String.<number>
CRLDistributionPoints	No	Distribution point (URL) of the CRL

TLS Client Certificate

When a vendor A's software wants to retrieve data from vendor B, the TLS connection is secured with certificates 2-way: vendor B presents a server certificate and vendor A with a client certificate which is issued by vendor B. That way, vendor B can easily authenticate incoming clients by testing them against its own certificate chain.

Issued certificates should contain the 'Nuts domain' extension, which can be used to determine whether vendor (A) is allowed to receive Social Security Numbers or that they should be masked. The 'Nuts domain' of the issued certificate should match that of the vendor's (A) Vendor CA Certificate.

Issued certificates should also contain the vendor's (A) ID as subject alternative name.

Extension / Field	Critical?	Value
Subject		CN=<Vendor A> <Environment>,O=<Vendor A>,C=NL
KeyUsage	Yes	digitalSignature
ExtendedKeyUsage	Yes	clientAuth
SubjectAltName	No	Vendor identifier, otherName: 1.3.6.1.4.1.54851.4=UTF8String.<number>
CRLDistributionPoints	No	Distribution point (URL) of the CRL
Nuts domain (1.3.6.1.4.1.54851.3)	No	UTF8String=healthcare | social | pgo | insurance

Note: The certificate is issued by vendor B to vendor A through an out-of-band process, meaning the network does not provide means to transport the certificate after issuance.

CN

The common name of the certificates used in two-way ssl must conform to the following specification:

> nuts-[network]-[app_name]

where *nuts* is static, *[network]* must be replaced by `development`, `test` or `production` and *[app_name]* needs to be replaced with the name also used in the login contracts.

Note: The specification of the CN might be changed to a certificate extension in the future, which will allow the CN to be freely choosen.

Note: The HTTP over TLS specification (RFC2818) mentions in section 3.1 that usage of Common Name is deprecated and that subject alternative names should be used. This only concerns the validation of a server's identity and does not imply that the use of Common Names should be avoided altogether. Therefore, we'll still use Common Names in our distinguished names without specifying it as subject alternative names as long as it's not a server certificate.

1.2 Consent classifiers

Inside the FHIR consent document there is room for specifying the extend of the consent under *provision.provision*. Current legislation only requires a simple yes/no for sharing data from a particular custodian/patient. Future legislation is uncertain.... To provide some type of scoping but still remain flexible when applying consent rules, **classes** are introduced. How and if specific resources at an endpoint fall under a class is to be determined by future rules and/or code. The benefit of a separate ruleset is that this can be made decentralized and (maybe) node specific. This will also enable the rules to be updated without having to release new versions of the software.

1.2.1 Available classes

Medical

The most interesting class, represents all that is medical: diagnosis, problems, plans, measurements, observations, medication, etc.

Social

All information regarding the social status, relatives and maybe the most important: other care providers.

1.2.2 Encoding

Different records require different encodings. The medical class encoded as single string value:

This encoding is usually the case in the Nuts REST APIs. The encoding as a tuple (system/code) as required in the FHIR records would look like:

```
"coding": [
  {
    "system": "urn:oid:1.3.6.1.4.1.54851.1",
    "code": "MEDICAL"
  }
]
```

1.2.3 Querying and checking consent

When querying consent for a specific *custodian-patient-actor* combination, classes will be returned. As stated in the beginning there are no rules yet for translating the classes to specific resources at the available endpoints. It is up to the individual vendor who represents the **actor** side to determine what to call when consent has been given.

The same goes for the **custodian** side. The vendor must check the JWT for authenticating the actor but also check given consent for authorization. The check must work the same as the query at the actor side of things. Currently it's

up to the vendors at both sides to determine which resources fall under a given class. When more vendors join and more use-cases are supported, this free format might have to be changed to some rules...

Query example

Using the *query* call from the *Nuts consent store API*, the request body would look like:

```
{
  "custodian": "urn:oid:2.16.840.1.113883.2.4.6.1:48000000",
  "actor": "urn:oid:2.16.840.1.113883.2.4.6.1:12481248",
  "query": "urn:oid:2.16.840.1.113883.2.4.6.3:999999990"
}
```

No surprises there, a return would be:

```
{
  "page": {...},
  "results": [
    {
      "id": "SOME_LONG_HMAC_ID",
      "actor": "urn:oid:2.16.840.1.113883.2.4.6.1:12481248",
      "custodian": "urn:oid:2.16.840.1.113883.2.4.6.1:48000000",
      "resources": [
        "TODO AFTER consent-store classifiers update"
      ],
      "subject": "urn:oid:2.16.840.1.113883.2.4.6.3:999999990",
      "validFrom": "string",
      "validTo": "string",
      "recordHash": "string"
    }
  ],
  "totalResults": ...
}
```

Which returns the **MEDICAL** class. With this response the requesting software must make a translation to specific resources, eg: the FHIR Observation resource or FHIR Patient resource.

In the future, a switch to POST search calls can be made to prevent identifiers leaking into access logs.

Check example

The check has to be done using *official* identifiers codeable in FHIR requests and Nuts consent records. The Dutch BSN, for example, is coded as:

in a FHIR request, and as

in Nuts records. Any code making the check must be able to do this translation. Nothing has been standardized yet, but a mandatory inclusion of the same identifier (may have different coding systems) in the FHIR call as well as the Nuts consent record might be the smart thing to do.

A *check* request body would then look like:

1.3 Distributed Privacy Consent

Throughout the documentation we'll be talking about *DPC* records. A *DPC* record is an unique record representing multiple consent records and parties with the same:

- **Custodian** The care organisation responsible for managing the data. This is typically the party with which the patient has an agreement for providing care.
- **Subject** The patient, client or civilian. Any data transferred between parties is about this subject.
- **Actor** The care provider that is granted access. The actor needs the data in order to provide better health care.

1.3.1 Reasoning behind DPC

The main reasoning behind a *DPC* record is that a care provider (actor) wants to access some piece of data about a patient (subject) from a different care organisation (custodian). In the Nuts network model, the software from the actor will connect directly to the API from the custodian and pass along the subject in the request. The actor uses an Irma signature for authentication and also passes along the identifier of the custodian. With this information the authenticating API endpoint has enough information to consult the consent records and make a decision.

Note: In a SaaS environment, multiple custodians can be accessible through the same API endpoint. Passing along some identifier for the custodian is then needed.

A *DPC* record also maps very well to the health care process itself. Asking a subject if information about him may be transferred to other software systems is ridicules. Asking if a particular care provider can get access on the other hand is what is currently done today.

Note: A General Practitioner might ask the following question during a consult: "Is it ok if the pharmacist can access medical information from this office?"

It would be an option to exclude the actor from a *DPC* record (and make it a Subject Domain Link) and just store everything about the subject in a single record with lots of attachments. This would mean though that all care organisations and/or care providers are involved parties in any transaction. It would no longer be possible to isolate a certain part of subject's health care network, which is unacceptable based on the manifest of Nuts.

Note: Without the actor in a *DPC* record, a mental health care professional would be an equal party to, for example, the home care organisation. Both organisations would be able to 'see' each other as parties where active consent is registered. Even if the home care organisation does not have consent to access the data from the mental health care organisation it would still know some care is being given. Given some extremely specialised care organisations, it would then be possible to determine what kind of care the subject is receiving. FilteredTransaction also won't work since the participating nodes are still known, node names can give away the care organisation.

1.3.2 FHIR mapping

Consent in Nuts is stored as encrypted FHIR consent records. Next to the three parties involved in a *DPC* record, the FHIR consent record also mentions:

- A witness, the person who has witnessed the consent agreement.
- The Consentor, the person who has given consent on behalf of the subject.

In many cases, the witness will be a care professional who has witnessed a verbal agreement or uploaded the wet-autograph as a user of the system of the custodian. In the case of a digital signature, the witness is a less crucial part of the consent record. The consentor can either be the subject itself or anybody that has a mandate to give consent on behalf of the patient. These two parties are not present in a *DPC* record since they only say something about the validity of the record and not the uniqueness.

1.3.3 PGO and implicit access

The above sections all use the professional to professional data transfer as subject. The patient himself via his PGO (Personal Health care Environment) is also part of a *DPC* record. The PGO though is not an actor and not a custodian, it represents the subject. So given a *Triple*, a PGO can be added as party to the consent record when it represents the patient. Any request a custodian API receives from a PGO user must therefore check the consent records for the patient identifier and not the actor. This way the patient can view and change the consent records as a party directly involved. For a PGO to view any medical data Nuts supports the notion of a *Subject Domain Link*. Viewing any medical data in a PGO must be possible regardless of any consent records being available. A patient must be able to view their own data even without any consent registered. Therefore the *Subject Domain Link* is a different structure than a *DPC* record.

1.4 Event sourcing and CQRS in service space

Creating, accepting and finalizing consent requests is a lengthy and complicated process. Besides the steps an individual node needs to take, nodes must also be aware of each other. Without this knowledge it's hard to give a user information about the state of its request. From a technical point of view, there's also the complexity of retrying failed steps and disaster recovery. Complex problems in their own definition, but even harder when let lose in a distributed environment where acceptance of the technology depends on simplicity.

Enter event-sourcing, a technique that stores events in a log, ready to be re-run at a later stage. The example most commonly used is that of a purchase or order, where the order goes through different states before being completed. Consent rules can be seen as *orders*. Together with event-sourcing, CQRS is commonly used in these types of systems. The main idea is to make the entire system event-based, where different services react on different events. Events are handled by multiple services. One of these services is responsible for creating the current state. This state can then be queried by vendor-space for the current state of a consent request. Other services act on the events to help transition the state. In the case of *Disaster recovery* the entire event log can be replayed.

1.4.1 Error handling & retry

A single fully accepted consent request has a 100 different ways of failing on its route to acceptance. Many of which can be a result of temporary problems such as communication problems and/or unknown technical problems. Some might occur through incorrect configuration and some might occur due to version conflicts. Almost all of these problems can be fixed by retrying at another time. Only functional rejected requests have to be dealt with by a user.

Communication problems occur because of network hiccups or external services that are down. Hiccups can usually be fixed by retrying processing immediately. External services that suffer downtime are usually fixed within a minute when a hot-standby HA method is chosen or within 15 minutes if an administrator has to deal with it. In badly maintained environments this can be stretched to several hours.

Problems that occur due to misconfiguration, like an incorrect address should be spotted right after an update or after installation and are expected to be fixed within an hour.

Version conflicts can occur when another node is running a newer version than the node giving the error. In Nuts, this is avoided by updating the internal processing and data model in a version before any interfaces are changed. This means that nodes can handle newer version before they are actually used. Two major versions are allowed to be active at any given time in the network (allowing for a transition time). If for some reason a node falls two versions behind

and is thus unable to process newer events, a retry in a week or so would fix this, since this will probably only occur in test networks. A test network will be updated quicker than a production network.

The retry mechanism is implemented as a single channel of events per retry interval. So 1 channel for 1 second retries, 1 for 1 minute and so on. The increase in interval and the maximum number of retries is configurable. The recommendation would be to do something like: 1s, 1m, 15m, 1h, 24h, 7d. The retry channels/queues are not durable, meaning they won't survive a restart of the node. When restarted the *Disaster recovery* mechanism will retry any non-completed events anyway.

An event that results in an error condition is *acked* and then published to the correct retry channel. Failing to publish to the retry channel (or to the event log in general) is a **Fatal** error.

1.4.2 Encryption

Because the events hold the actual consent data (and thus personal information), it can only be stored to the event log **after** it has been encrypted. All events are processed in-memory except the event log. Encrypting the data before the first event makes sure that records written to disk are encrypted. This means that the first REST call for creating a new consent record will be synchronous until the encryption step. After encryption it'll be an event and the event UUID can be returned to the caller.

1.4.3 Disaster recovery

In the case all Nuts nodes have failed and the entire system has to be started again, the event log can be used to resume consent requests. This is done by first starting the event store (nuts-event-octopus module) which will re-read the entire event log. This will recreate an internal db (SQLite) with the current state of each consent request. Any vendor-space services can now already happily query request states. After the event store has been recreated, the other modules will be started that are depended on events. Each event is processed by each module. Modules only react to their events they are interested in and they will check with the consent store if the given state is still current, if not the event is skipped.

Note: An event log is a ring-based log and therefore has a maximum size. This size has to be chosen wisely, for normal operations it can be expected that at most 1 consent request per second is handled. Each request has to go through 6 states or so. So for storing a day of logs, a log size of *6 * 24 * 60 * 60 = 518400*. This will account for downtime of the entire nuts network for a full day. This is highly unlikely, these log sizes do, however, are needed when doing bulk imports. For example, a hospital with 100.000 active consent records, will need a log size of 600.000 to do a full bulk import. (probably less, but better safe than sorry).

Note: In the future the last processed event ID can be persisted, so modules can skip the first X events.

1.5 Application flows

Application flows are the main flows of interaction between vendor application software and the different parts of the Nuts nodes. The paragraphs below will explain what happens internally and why things work as they do. There are two main flows: the consent flow and the data flow.

1.5.1 Registering consent

The diagram below shows the flow between two parties. The *actor* is the party that wants to request some data about a patient and the *custodian* is the care organization holding the data. Both parties have their own information system (*XIS*) and both are running a Nuts node within their infrastructure. The Nuts registry has been omitted from the flow, but is used internally by the node for translating an organization identifier to endpoint.

The flow starts at the top-right of the diagram (yes we like to be different) by a user registering consent via a UI. The XIS is responsible for contacting the Nuts node via the *Nuts consent logic API* which starts the *Create Consent Branch* flow. The result is a handle to the transaction which can be used to follow the state of the transaction. It's unwise to make this a synchronous process since there's no limit on how long the transaction can take.

All nodes that are involved in the transaction must validate the transaction. The transaction must be validated technically (syntax, references, timestamps) and contextually. Each node must validate that the *BSN* indeed belongs to a patient registered in the local XIS (due to legislation, ID card check and such) or will be in the near future. If a patient is not in care at one of the nodes, the transaction will be cancelled. An *ack* by a node is represented as a *Sign Consent Branch* flow in Corda.

Note: When a node starts too many invalid transactions, it might be an indication that the specific node is misbehaving and must be isolated in the network. A symptom might be a lot of negative responses to the *patient_in_care()* call.

Transaction validation by a node is done by adding a signature. In the diagram this is shown by the *sign()* call and *Sig* abstraction. The initiating node does this as well, but most likely does this in an automated way. The other nodes can choose to do it automatically by searching for a patient (by bsn) in their own records or manually validate it through an UI. For most of the implicit consents, a patient record will not be available yet at the *XIS actor* part. In that case the validation will most likely happen through an UI where the consent request can be linked to an out-of-band communication effort.

The *ConsentBranch* synchronization in the flow diagram is handled by Corda.

When all nodes have validated the transaction, the initiating Nuts node will finalize the transaction (*final()*) by calling the *Merge Branch* flow. A result will be a synchronized state at all involved nodes. The Nuts node can emit an event for the XIS to notify users or update some state.

1.5.2 Data retrieval

Warning: The security for the data request is currently in flux. If the *works-for* relation has to be proven as well, this will no longer fit in http headers. Also when additional context like a patient identifier or a specific task number is required at the custodian end, you don't want the token to be created at request time. Therefore an access token will be used. The token can be requested by an out-of-band call, similar to Open ID Connect, Saml and OAuth2. This documentation will be updated when available.

The diagram below shows the flow for retrieving data. Most important to note is that the Nuts nodes are not responsible for the data exchange.

An actor requests data from another system from the XIS it's working in. The XIS needs the identity of the actor before the request can be made. The Nuts node has convenience services for generating a QR-code which can be scanned by the IRMA app. The resulting signature/token can be used in the data call. This can be done by calling the correct nuts-auth-api call. Specification on the IRMA signatures can be found at *Login Contracts*.

Given the identity of the current user and the context of the current care organization, the Nuts node can be used to query available consent by calling the *Nuts consent store API*. This can be done with or without a patient context. The result will be a class of data that can be retrieved for custodians. The translation of class to resources is future work...

Given a custodian from the resulting consent API call, the technical endpoint can be retrieved from the *Nuts registry API*.

Now all requirements are met for doing the data request (endpoint, user identity, actor identity, XIS identity, patient identity, custodian identity).

The data request that arrives at the custodian XIS endpoint will have to be validated in the reverse order is sort of the same manner:

- is it a secure connection?
- are the given identities valid?
- has consent been given?

The later two can be checked by two API calls on the Nuts node. The return values are nothing more than a yes/no response. More details can be found on the API pages: *Nuts consent store API*, nuts-auth-api and how to implement it: *Let users authenticate themselves to the Nuts network*. The two-way TLS connection will be established with vendor specific certificates coming from a CA specified in *RFC: Distributed Registry*.

Note: Some more references to other pages to add when they come available.

1.6 Subject Domain Link

Next to a *DPC record*, Nuts also has a *Subject Domain Link* consent record. This record represents the *link* a PGO (Personal Health care Environment) has to the medical domain. A *Subject Domain Link* represents the following parties:

- **Custodian** The care organisation responsible for managing the data. This is typically the party with which the patient has an agreement for providing care.
- **Subject** The patient, client or civilian. Any data transferred between parties is about this subject.

1.6.1 Isolation

A *Subject Domain Link* is created every time when a user/patient makes a link with the medical domain by disclosing their BSN at the inclusion website of a care organisation. This will let the *service space* of that care organisation know that a particular PGO user may be linked to a BSN record. The uniqueId for the *LinearState* will be created as an HMAC_256 of BSN and PGO user id. The user can choose to make this *link* private or *viral*. When private, the user can only access data for that particular care organisation. Other care organisation have to be added in the same way by creating new *links*.

1.6.2 FHIR mapping

The *Subject Domain Link* also uses an encrypted FHIR consent record as data carrier. The policy will most likely be simpler since it'll only target the patient role itself.

1.6.3 Subject Domain Links going viral

When a user makes a *link*, it should be able to choose it to be able to go viral. This means that the care organisation is allowed to send the consent record to other parties that are involved in the care of the patient. The parties receiving the consent may then also do the same. This 'register once, see all' principle will greatly reduce the burden for the user.

When the viral option is chosen, two consent records are attached to the *Subject Domain Link*: the actual consent from the subject to the custodian and the consent records allowing custodians to redistribute the first record. The records are limited for care organisations only. This would block any Nuts node from sending it to a non-medical Nuts node.

The image below is a first attempt to identify all the required components for making Nuts work. All interfaces between components will have a specification. Each vendor has the choice to implements their own components following the spec or just use the components provided by Nuts (mix-and-match).

At first glance, the different components are divided into four different *spaces*. This is done to distinguish between different levels of trust and (probably) different non-functional requirements like scaling. *Vendor space* has the highest level of trust. It includes both personal and medical data. Any components required to make Nuts work **must** be implemented by the vendor. *Service space* is also a trusted space since it'll include decrypted personal data, but not medical data. Since it consists of quite a few components a vendor can buy this as part of a service package from a different vendor. For example, a simple PGO might want get a subscription to a SaaS service which includes both *Service space* and *Nuts space* which only requires the PGO to implement the *Patient callback*. *Service space* also handles the administration of all the certificate logic for TLS connections. *Nuts space* only contains encrypted personal data and decrypted personal data that has already been made public. The primary goal of *Nuts space* is to make sure all required data for running the Nuts network is distributed across all nodes without any centrally controlled component. This notion of different service levels allows smaller vendors to connect to Nuts and at the same time provide an opportunity for the current network providers to develop new business. *Nuts foundation* is a separate entity which has the responsibility for adding the nodes. Essentially it controls the network. Although we don't like centrally controlled components, a central authority or root is required. In this case the *Nuts foundation* will control the root certificate. This is a direct result of the available technology, in the future other technology might become available or with enough Nuts participants, we can create something of our own.

Communication between the different components is done via REST services and ZMQ. In the future different protocols might be supported. Different security measures are/will be supported between the different components.

1.7 Vendor space

Vendor space contains all the components that **must** be implemented by each vendor with exception of the PGO UI. Most of the components will already be available since they are part of the EHR. Each of the components will be explained further in the following sections.

1.7.1 Public EHR API

The *Public EHR API* is the primary point of data exchange for a care provider. It'll be based on international standards, for example: FHIR or CDA. This component is placed in vendor space because the underlying TLS connection will be secured with two-way SSL. The control over the certificates should lie with the entity responsible for the data: the care provider. Ultimately, it'll be the software vendor who will probably offer this responsibility as a service.

The *Public EHR API* will most likely consist of multiple components in an average production environment. The main responsibilities are:

- Authentication
- Consent authorisation
- Auditing
- Return data in requested format

Authentication

Authentication within Nuts will be provided by an Irma signature. The signature will have a signature text and attached attributes used to sign the text. Within the signature text, the requesting application name, the requesting care provider, the goal binding and a timestamp must be included. The attributes used to sign the text will identify the user. The requesting application name will also be in the CN used in the two-way TLS connection, this ensures that the care provider/vendor receiving the request can not use the same signature to request the Nuts network (man-in-the-middle). The timestamp ensures a short lived session. All the other information will be used to lookup the required consent record.

Consent authorisation

The consent check is basically the authorization check for the request. Every consent record basically consists of a quadruple: (care provider, patient, care professional, scope), or 'May the care professional access the specific scoped data for the patient managed by the care provider?' The patient can be acquired by analysing the request, for example within FHIR the BSN will be included in the request, the care professional can be determined by the attributes used to generate the signature. The query to the consent registry within Nuts will then result in a list of care providers which can be used to query the correct data (or not found). Access to the specific resource is determined by the scope of the consent.

Data

If all previous steps have succeeded, the API component can then proceed to fetch and convert the data as requested. The initial versions will probably use FHIR as standard of choice.

1.7.2 Patients DB

This *database* represents the patient's personal information and specifically the BSN record. All EHRs will contain this information since it's required by law. The data needs to be accessible by the *patient callback* component.

1.7.3 Patient callback

This component will mainly be used to check if a patient is receiving care for the given care provider. If, for example, this care provider receives consent to access certain patient records from another care provider it can only accept this consent if the patient is really a patient there. If not, the BSN may not be stored and the consent request can not be accepted. This check can also be used to detect faulty or corrupt Nuts nodes, since a lot of negative results from this component may indicate fraud. In a later version of Nuts this can be used for automatic blacklisting of Nuts nodes.

1.7.4 EHR UI

The EHR UI represents the piece of software the user interacts with. The part that is particular interesting is the consent UI. In the early stages of Nuts, the care providers will probably do all the work for gathering the patient consent. This means that the EHR needs to have a UI capable of recording this consent.

1.7.5 PGO UI

This component represents the UI needed for the PGO-inclusion flow. An idea exists where a patient is redirected by a PGO to this component to link their PGO identity to a BSN. The vendor can then use the Nuts network to update the consent records with the added PGO identity for all existing consent records for that patient. The UI needs to be

in vendor or service space, otherwise the BSN can not be used. The difference between putting it in vendor or service space would be if it's embedded or not. Nuts will provide a reference implemention for placing it in the *service space*.

1.8 Service space

1.8.1 Consent store

All consent within *Nuts space* is encrypted. The store will have a unencrypted copy of the records in memory to support querying from, for example, the *API*. The attached *encrypted storage* will ensure that this sensitive data is encrypted-at-rest.

1.8.2 Consent Logic

The logical component does most of the heavy lifting and depends on all the other components in *service space*. For example, when creating a new consent request, this is send to the component it then checks if it's valid by using the validation component. Next it has to find the correct organizations and encrypt the record with the right public keys. Then it has to send the encrypted record to the Consent bridge for synchronization.

Also when a new consent event is received by the component from the consent bridge, it needs to decrypt it and check its validity. If valid it has to be send to vendor space to check if the subject is really a patient for that care organization.

1.8.3 Consent validation

This component handles all logic regarding validating the FHIR consent record. It checks the content via different rules predetermined by Nuts.

1.8.4 Crypto

The crypto component is an abstraction layer for the encryption/decryption process and the storage for pub/priv key-pairs. The abstraction is needed to support the different use-cases. A PGO might choose for file-storage since it'll only have a single key-pair. A service provider might choose for a Vault installation because it handles thousands of keys.

1.8.5 Irma

Generic Irma server for checking Irma proofs.

1.8.6 Nuts auth

This component is responsible for checking the different Irma signatures used like login and connect (PGO). It connects to the *Irma* server for checking Irma proofs if those are used to sign a consent record. This can't be done in Nuts space since it will then be encrypted.

1.9 Nuts space

The *Nuts space* consists of two main components: *Consent Cordapp* and the *Nuts registry*. The other components are requirements coming from technology choices for these two components. The funky figure within these components

indicate that they use distributed technology. They basically are a data store without a single owner and the single truth is constructed from mutual approved contracts.

1.9.1 Nuts registry

The registry contains mostly relational and identifying information. It must be able to answer questions like:

- What is the FHIR endpoint for this care provider?
- Which Nuts nodes serve a particular Care Provider?
- To which care provider does this care professional belong to?
- and others

The consensus about the data is constructed by a few different rules:

- It'll probably contain a tree structure, where a lower level node can only be **added** by a higher level node.
- Only the **owner** of a piece of data can update that data.

Which can be translated to things like:

- Only a Nuts node can add a care provider/application/service to a that Nuts Node.
- A care professional can only be added to a care provider by the care provider.
- The personal data of a care professional can only be updated by that care professional.

To guarantee these constraints, cryptographic rules have to be used. Nuts will probably use a combination of Irma signatures and digital signatures (PGP) for this.

Since the data within the registry is useful for everybody using Nuts, it can use a mesh network to keep in sync.

1.9.2 Registry UI

There'll probably be two UI's: one for administrative purposes and one for care professionals to update their information. The last will then probably be a reference implementation provided by Nuts, since vendors can offer such an interface from within their own products.

1.9.3 Consent Cordapp

The *Nuts Consent Cordapp* (What is a Cordapp?: https://docs.corda.net/cordapp-overview.html) is responsible for creating the decentralised state of consent. The *Corda Consent Model* therefore consists mainly of encrypted data. Validation of any data specific constraints will be delegated to *Service space* during a Corda transaction.

The Corda node which will store all the consent records. Corda has currently been chosen to store the consent. It's unique ability to only include nodes that are part of the consent in the transaction makes it ideal to synchronize personal information. Although the data itself is encrypted, having it all over the place just isn't a good idea. Another plus is that it requires a third party to also acknowledge the transaction (the notary). It can even use a voting scheme to include multiple random notaries. This means that the control over all transactions lies with the community and not a single party.

For every transaction, each involved node needs to approve the transaction according to the logic in the contract. This will rely on data available in the *Nuts registry* or even the *patient callback*, proxied through *service space* for decryption. This will prevent data to scatter all over the place.

1.9.4 Consent bridge

The goal of the *Nuts Consent Bridge* is abstract away from the Corda specific classes and logic. It also exposes logic and data language agnostic. Corda is written in Java/Kotlin. The *Nuts Corda Bridge* emits events using NATS. NATS has libraries in many languages. There are two main interfaces on the bridge: the publish/subscribe endpoint and the request/response endpoint. Check nuts-consent-bridge-technical for more info.

1.10 Nuts foundation

The *Nuts foundation* controls the root certificate, defines which nodes are added to the network and which versions of the Cordapp are allowed. This is needed because Corda requires a CA tree structure. Corda also requires a NetworkMap which must be signed by a single key. The control of this key must lie with a trusted third party. This party can only accept/reject Nuts nodes, it cannot exchange medical or personal data.

1.10.1 Nuts registry

The *Nuts foundation* will also run a *Nuts registry* instance to add the Nuts nodes so they can be found by other nodes. The Nuts nodes can then add new organisations themselves.

1.10.2 Nuts Consent Discovery

The *Nuts Discovery Service* is the Nuts implementation of the Network map service described by *Corda*. The *Corda* specific documentation can be found at https://docs.corda.net/network-map.html. The reason for implementing the network map as a service and not just distributing node information via other means is that this greatly simplifies development, puts the control of the root CA at the right place and creates a bridge to the *Nuts registry*. When a node registers with the discovery service, the service can also add the node to the registry. This will enable node administrators to link care providers to their Nuts node entry in the registry.

Corda often speaks of the *Doorman*. This is the *service* that is responsible for approving nodes, eg: signing certificate requests. The *Doorman* uses the intermediate CA to sign Node CA's. Right now, Nuts combines the *Doorman* service and the *NetworkMap* service in the *Nuts discovery Service*.

CHAPTER 2

Technical documentation

2.1 Consent Cordapp Overview

The key concepts of Corda are States, Contracts, Transactions and Flows. The chapters below gives a high level overview of how they are related. More information can be found at The corda documentation.

2.1.1 States

A Corda state represents a current state of a particular record. Everything in Corda is immutable so states can only linked together by referencing the previous state. States are stored in the vault together with any attachments. Corda has different types of states, within Nuts we use the LinearState. A *LinearState* has an *externalId* that never changes, this will be the link Nuts needs to connect encrypted records to real patient records. The *externalId* will be created by the Party that creates the *LinearState*. The *externalId* will get a unique constraint in the underlying database. Because of the encrypted records, this id is the only way to prevent mass-duplicate records. In order to achieve this, the id must be unique but also reproducible. It therefore must use some sort of consistent hashing algorithm like *HMAC_256* using the private key of the organisation. This will only prevent duplicate records initiated by a single node. This will not prevent duplicates from multiple sources when initiated at the same time. Although it would be nice to prevent this as well, it's extremely hard to prevent and will almost never occur. This can be fixed by merging two records. Duplicates are only a problem with the initial record, if a record already exists, any change is an update and is therefore regulated by a notary.

Important:

The Nuts Corda Consent model represents *DPC records*:

- The care provider responsible for storing the data (the custodian)
- The patient (the subject)
- The care provider that is granted access (the actor)

Together with states Nuts also stores attachments. The attachments hold the actual data and are encrypted with a symmetric key. The symmetric key can only be accessed by the organisations that are involved in a *DPC* record. To ensure privacy, the symmetric key is encrypted with the public key of the involved organisations.

Note: Next to the two involved care providers (custodian and actor), the PGO (Personal Health Environment) can be an involved party as well. When this is the case, extra attachments will be present containing the patient identifier for the PGO. The attachments that reference the official patient identifier (BSN) will be hidden for the PGO.

More information on the structure of the attachments can be read in the *Corda Consent Model* chapter.

2.1.2 Contracts

Contracts in Corda are the pieces of software that check state transitions. Given a particular action a state will be consumed and a new output state will be created. A contract makes sure this operation is atomic across all participating parties. Since Nuts consent states only have an *externalId* and attachments, the contracts will mainly look at the attachment data to make sure that certain constraints at the meta level are correct. The *Nuts Consent Cordapp* will store the state of a consent request before making it an actual *DPC* record. Before the final *DPC* record can be created the other node has to check the attachment. Because this is a potential long-running operation it needs to be stored as a separate state. The attachment validation is done in *service space* (Under architectural construction).

2.1.3 Flows

Flows are the main pieces of logic to create or update states. They are basically a step-by-step description on information gathering, sending data to other parties and finalising the transaction. Flows are distinguished into sending and receiving flows and can contain any number of sub-flows. The sub-flows allow for reuse of functionality. The current supported flows can be found in the *Supported consent flows* chapter.

2.1.4 Transactions

A Corda transaction is the process were input states are transformed to output states according to a flow. All involved parties will sign the transaction if it adheres to the contract associated with the transaction. When all parties agree, a notary will sign the transaction making it final. The parties will then proceed to update the changes in the vault. No custom transaction logic is possible and this is entirely handled by Corda.

2.2 Corda Consent Model

When we talk about the *Corda Consent Model* we mean the *LinearState* and all attachments involved. This model has little actual data (only encrypted data) and is only used to persist a state across parties. Any constraints involved are to make sure encrypted data only arrives at the places it's meant to.

Note: Even though data is encrypted and therefore it'll not be regarded as a data breach (GDRP) when leaked/stolen. It's still wise to not send it all over the place. Because when the used encryption algorithm is deemed unsecure, that old data that was send to unauthorised parties now suddenly becomes a data breach! And since data is immutable and will be stored for at least 15 years, nothing will be certain...

2.2.1 Attachments

Corda attachments are basically several files bundled in a single zip-file. They are not part of the transaction, but their hash is. It's cryptographically hard to change a file so that it has the same hash as a different piece of data (not enough energy in the universe type hard). Attachments can also be large since they'll contain an encrypted version of a FHIR bundle. Not having to send the data all the time can be a time saver. *Custom flow logic* is responsible for sending or requesting different attachments from another party within the same transaction.

Attachments referenced in a Nuts consent transaction contain two files:

- cipher_text.bin
- metadata.json

The *cipher_text.bin* file is the encrypted FHIR consent bundle and the metadata contains data on how to decrypt the file and three custom fields:

- domain, denotes the target audience for this attachment: medical, PGO or insurance nodes
- previousAttachmentId, points to the previous attachment if the FHIR consent model has only be updated (increase of consent resource version)
- period, the only part of the FHIR consent record that is also available without encryption. This is done to prevent sending attachments to other parties when they have already expired.

2.2.2 metadata.json schema

meta.json root			
type	*object*		
properties			
• **domain**	from domain code-system		
	type	*array*	
	items		
	•	type	*string*
		enum	medical, pgo, insurance
• secureKey	type	*object*	
	properties		
	• alg	Symmetric encryption algorithm (eg 'AES_GCM_256')	
		type	*string*
	• iv	Base64 encoded initialisation vector	
		type	*string*
• organisationSecureKeys	type	*array*	
	items		
	•	list of encrypted symmetric keys. Uses the organisation public key to encrypt	
		type	*object*
		properties	
		• organisationId	URI representing the id of the organisation according to Nuts naming schemes
			type *string*
		• alg	Asymmetric encryption algorithm (eg 'RSA_3k')
			type *string*
		• cipherText	Base64 encoded cipher text representing the symmetric key

Continued on next page

Table 1 – continued from previous page

			type	*string*
• previousAttachmentId	Refers to the previous version of the encrypted consent record. Can be used to recreate history.			
	type	*string*		
• period	indicates the validity period of the encrypted consent record. The period must be the same as the encrypted one. This data can be used to 'garbage collect' attachments and keep the transactions as small as possible			
	type	*object*		
	properties			
	• validFrom	Inclusive start date		
		type	*date*	
	• validTo	Exclusive end date		
		type	*date*	

2.2.3 Attachment lifecycle

Attachments are immutable but the data they contain is not. The underlying model can be updated or even additional consent records can be created for the same *DPC* record which can be valid at the same time. This problem can be solved by using multiple attachments per *DPC* record.

Note: A *DPC* record (Custodian, Subject, Actor) is unique but multiple consent records can be active at the same time. For example a patient can allow the hospital to access the involved care providers list present at a home care organisation. At some point in time the patient might be taken into the hospital for some treatment. Next to the list of care providers, the patient might want to allow the hospital to access the medical records at the home care organisation, but only temporary. An additional consent record will have to be present, which is only valid for a certain time, next to the 'base' consent record.

Only two actions are possible for consent records: *create* and *update*. A deletion is, in essence, just an update where the end date will be set to a certain point in time. For auditing reasons the entire history must be kept as well. When a consent record gets updated, two new attachments are created for that transaction: the old consent record but now updated with an end date. The underlying FHIR model will keep its ID and version number, and a new consent record valid from today onwards. This underlying FHIR model for this new record will have the same ID but with an increased version number. The updated record will also point towards the previous attachment so any party can view the history of the record.

2.2.4 Request/Accept state

The creation or change of a *DPC* record is not a single transaction. Before a *DPC* record changes, a new `ConsentBranch` needs to be created for the *DPC* record. The involved parties will listen to these new states and will validate the attachments that come with it. When the attachments are validated, a new transaction is started by the receiver to accept the new request and update the *DPC* record. A big challenge with this flow is that the validation happens in *service space* and that the consent contract can not access the Nuts registry or validate that a certain public key is indeed the public key of a care organisation. Validation of public keys in service space is sufficient since all encryption is connected to the public keys from the registry. So any spoofed identities or wrong public keys will result in faulty encryption. In the case a `ConsentBranch` is wrongly accepted and finalized by a malicious node, other nodes will still log an error on processing the *DPC* record since its public keys will be unknown. This will also prevent those nodes from decrypting and storing the *DPC* record.

2.2.5 Additional parties

It is possible that a single care provider or organisation is using multiple pieces of software. Something which is common if an organisation adopts the best-of-breed approach in software selection. In that case you don't want to register additional consent. The way Nuts deals with this, is that a care organisation will have a unique set of keys which can be exported and uploaded to *Nuts service space*. When a new care organisation is registered for a Nuts node in the *Nuts registry* other nodes can act on this event. If another node identifies the care organisation as a party that is registered at that node, it can pro-actively add the new Nuts node as a party to the existing consent records. The same mechanism can be used for migration purposes as well.

For the consent cordapp this process is an update of the state but without any attachments changing or added/removed. Only an involved party is added and all parties sign the transaction. All checks still need to be done by all involved parties.

2.2.6 Filtered transactions

Corda supports a concept called 'transaction tear-off' or 'filtered transactions'. This allows flows to hide certain data for certain parties. The transaction is still valid because the signing of the transaction by all the parties is done by signing a hash of the actual data. This allows parties to see and sign a hash without seeing the data. The entire transaction is constructed as a Merkle tree, where parts can be substituted by their hash.

This concept is used by Nuts to hide the BSN (Dutch national number) from PGO's (Personal health environment) and still allow the PGO to be part of the transaction.

Note: The choice to let the PGO be part of the transaction is an important one. There's no shadow bookkeeping happening in order to distinguish between different environments and requirements on identifiers. Because the PGO and therefore the patient is part of the consent state, it'll always have the latest information on who can access their data!

2.3 Nuts consent discovery

2.3.1 Keys and certificates

The figure at https://docs.corda.net/network-map.html shows the CA structure of Corda nodes. The *Nuts Discovery Service* represents the *security zone* in that figure. To startup the service, a few keys need to be generated and configured. More info on this can be found in nuts-discovery-setup. Only the root CA is configured at each node as trusted anchor. Both the Doorman and Network Map CA are trusted because they are signed by the root. The CertRole for each CA is important, to prevent mistakes configuration files for generating these certificates can be found in this repository.

Note: All mentioned keys and certificates are only used for the Corda part of the Nuts network. Communication between care providers use different certificates. This ensures that only the right parties can issue/revoke certificates based on the responsibilities they have. The *Nuts Discovery Service* can only determine who joins the network. It cannot access or manipulate any data. The root key should not be present on any production environments. It can even be controlled by a different part of the Nuts community to ensure decentralization of control.

2.3.2 Modes of operation

The *Nuts Discovery Service* currently has only 1 mode of operation: local development. In the future it will support production and other modes.

Local development

The local development mode is targeted at running a Nuts network locally. To prevent any manual configuration, the *Nuts Discovery Service* will auto-sign all certificate signing requests it receives. Check nuts-discovery-setup on how to start the service.

2.4 FHIR consent requirements

This page lists the rules for the fhir consent model.

2.4.1 Minimal rules

The list of rules below will only give a valid json record. It will not be a valid Nuts consent record (yet)!

- The `resourceType` is equal to **Consent**
- `scope.coding` has an entry with `system` equal to **http://terminology.hl7.org/CodeSystem/consentscope** and `code` equal to **patient-privacy**
- `category` has a single entry where `coding` has an entry with `system` equal to **http://loinc.org** and `code` equal to **64292-6**

```
{
  "resourceType": "Consent",
  "scope": {
    "coding": [
      {
        "system": "http://terminology.hl7.org/CodeSystem/consentscope",
        "code": "patient-privacy"
      }
    ]
  },
  "category": [
    {
      "coding": [
        {
          "system": "http://loinc.org",
          "code": "64292-6"
        }
      ]
    }
  ]
}
```

2.4.2 Additional rules

In order for the Nuts components to use the consent records for validation, additional properties are required:

- `meta` is required
- `patient` is required and refers to a patient.
- `dateTime` is required
- `performer` is optional and refers to the user recording the consent.
- `organization` is required and refers to the custodian of the data (the organization).
- `source` is required and refers to the proof that has been given by the patient.
- `verification` is required and refers to the person that gave consent.
- `policyRule` is required
- `provision` is required and defines the extend of the consent.

Each complex requirement is explained in sub sections.

Meta

The `meta` field is used to track changes to the consent and might indicate there are previous versions. Only the latest consent proof will be referenced form the active document. If a later proof constrains the active consent, eg: end it on a specific date. Then the latest proof will point to the document proving the consent has ended. The `meta` field will then indicate that the current record has a `versionId > 1`. Previous records will still contain the proof wht consent has been given in the past. The `versionId` field starts at `1` and is incremented with *1* for each update. The `lastUpdated` field is also required and will indicate the last moment the record was updated.

Patient

`patient` Reference is required and the `identifier` field is present. `display` Is not allowed since no personal data is stored. The `system` of the identifier must be a valid Nuts system. In the case of patients this must be **urn:oid:2.16.840.1.113883.2.4.6.3**. This will be extended in the future when the PGO case is added.

```
{
  "resourceType": "Consent",
  "patient": {
    "identifier": {
        "system": "urn:oid:2.16.840.1.113883.2.4.6.3",
        "value": "999999990"
    }
  }
}
```

Performer

`performer` Refers to the user that initiated creation of the consent record. In the first stage this will be a **Practitioner**, where the **patient** or **relatedPerson** will follow in a later stage. In the case the user does not have a valid Nuts identifier but acts on behalf of an organization, an **organization** is referenced. `display` must not be present in the reference.

Note: An extra check has to be added to determine that the performer works for/is the organization when a poor proof like pdf is given.

Note: In the case of verbal consent, the profile needs to be extended to identify that type of proof. When verbal consent has been given, a valid login contract must be present identifying the user.

```
{
  "resourceType": "Consent",
  "performer": {
    "type": "Practitioner",
    "identifier": {
        "system": "urn:oid:2.16.840.1.113883.2.4.6.1",
        "value": "00000007"
    }
  }
}
```

```
{
  "resourceType": "Consent",
  "performer": [{
    "type": "Organization",
    "identifier": {
        "system": "urn:oid:2.16.840.1.113883.2.4.6.1",
        "value": "00000000"
    }
  }]
}
```

Organization

`organization` Refers to the custodian of the data. This is required and must use a valid Nuts identifier as reference.

```
{
  "resourceType": "Consent",
  "organization": [{
    "identifier": {
        "system": "urn:oid:2.16.840.1.113883.2.4.6.1",
        "value": "00000000"
    },
    "display": "P. Practise"
  }]
}
```

Source

The `source` will always be an `sourceAttachment`. The source always points to the latest change in consent proof. The attachment can have a `contentType` and can have an `url`. There are several valid contentTypes:

- application/pdf
- application/json+irma

When the source is a pdf, it must be a scanned document with a wet autograph. When the source is of type **application/json+irma**, the data is the login contract of the *performer*. The title should reflect the type of consent given. Since no personal data is stored, the source only refers to a proof. The `url` must be accessible and must accept a Nuts identification method (eg: Irma signature in a JWT). The hash can proof the document has not been tempered with.

Initially the title will be the most important, when no online reference is available through an url, the title will be the reference clients/patients will use to contact the care organisation.

```
{
  "resourceType": "Consent",
  "sourceAttachment": {
    "contentType": "application/pdf",
    "title": "Toestemming delen gegevens met Huisarts",
    "url": "https://some.fhir.url/Document/1111-2222-33334444-5555-6666",
    "hash": "04298DE0...AB=="
  }
}
```

```
{
  "resourceType": "Consent",
  "sourceAttachment": {
    "contentType": "application/json+irma",
    "url": "https://some.url.domain/contracts/etc/file",
    "title": "Toestemming delen gegevens besproken met behandelaar"
  }
}
```

Verification

`verification.verified` should always be **true**, if **false**, the source should reflect this (eg. court order). `verificationWith` should refer to either the patient or a relative of the patient.

```
{
  "resourceType": "Consent",
  "verification": [{
    "verified": true,
    "verifiedWith": {
        "type": "Patient",
        "identifier": {
            "system": "urn:oid:2.16.840.1.113883.2.4.6.3",
            "value": "999999990"
        }
    }
  }]
}
```

PolicyRule

`policyRule` is either **OPTIN** with provision records or a general **OPTOUT** denying data to be shared from the given custodian. When **OPTIN** is chosen, `provision` is required to have at least 1 record.

```
{
  "resourceType": "Consent",
  "policyRule": {
    "coding": [
      {
        "system": "http://terminology.hl7.org/CodeSystem/v3-ActCode",
        "code": "OPTOUT"
      }
```

(continues on next page)

(continued from previous page)

```
    ]
  }
}
```

```
{
  "resourceType": "Consent",
  "policyRule": {
    "coding": [
      {
        "system": "http://terminology.hl7.org/CodeSystem/v3-ActCode",
        "code": "OPTIN"
      }
    ]
  }
}
```

Provision

`provision` holds the actual extend of the consent. It must at least have 1 `actor`. For now this must identify the **Organization**. The `role` will always be **PRCP**. `period` is required and has an optional `end`. `dataPeriod` is optional, when given it will restrict the data period for which data can be retrieved. `provision.provision` will hold all the specific resources that are covered by this consent. `type` is required and will always be **permit**. `action` is required and will allow for only **access**, **correct** or **disclose** (using *http://terminology.hl7.org/CodeSystem/consentaction*). `action` will list all the fhir resources that can be accessed (using *http://hl7.org/fhir/resource-type*). Nuts will also direct how a general consent category like *medical* can be translated to accessible resources.

```
{
  "resourceType": "Consent",

  "provision": {
    "actor": [
       {
         "role":{
           "coding": [
             {
               "system": "http://terminology.hl7.org/CodeSystem/v3-ParticipationType",
               "code": "PRCP"
             }
           ]
         },
         "reference": {
           "identifier": {
             "system": "urn:oid:2.16.840.1.113883.2.4.6.1",
             "value": "00000007"
           },
           "display": "P. Practitioner"
         }
       }],
     "period": {
       "start": "2016-06-23T17:02:33+10:00",
       "end": "2016-06-23T17:32:33+10:00"
     },
     "provision": [
```

(continues on next page)

(continued from previous page)

```
        {
          "type": "permit",
          "action": [
            {
              "coding": [
                {
                  "system": "http://terminology.hl7.org/CodeSystem/consentaction",
                  "code": "access"
                }
              ]
            }
          ],
          "class": [
            {
              "system": "http://hl7.org/fhir/resource-types",
              "code": "Observation"
            }
          ]
        }
      ]
    }
}
```

2.4.3 Complete example

The example below grants access to observations for Practitioner with agb=00000007 from patient with bsn=999999990 from organization with agb=00000000

```
{
  "resourceType": "Consent",
  "scope": {
    "coding": [
      {
        "system": "http://terminology.hl7.org/CodeSystem/consentscope",
        "code": "patient-privacy"
      }
    ]
  },
  "category": [
    {
      "coding": [
        {
          "system": "http://loinc.org",
          "code": "64292-6"
        }
      ]
    }
  ],
  "patient": {
    "identifier": {
        "system": "urn:oid:2.16.840.1.113883.2.4.6.3",
        "value": "999999990"
    }
  },
  "performer": [{
```

(continues on next page)

(continued from previous page)

```
    "type": "Organization",
    "identifier": {
        "system": "urn:oid:2.16.840.1.113883.2.4.6.1",
        "value": "00000000"
    }
  }],
  "organization": [{
    "identifier": {
        "system": "urn:oid:2.16.840.1.113883.2.4.6.1",
        "value": "00000000"
    },
    "display": "P. Practise"
  }],
  "sourceAttachment": {
    "contentType": "application/pdf",
    "title": "Toestemming delen gegevens met Huisarts",
    "url": "https://some.fhir.url/Document/1111-2222-33334444-5555-6666",
    "hash": "04298DE0...AB=="
  },
  "verification": [{
    "verified": true,
    "verifiedWith": {
        "type": "Patient",
        "identifier": {
            "system": "urn:oid:2.16.840.1.113883.2.4.6.3",
            "value": "999999990"
        }
    }
  }],
  "policyRule": {
    "coding": [
      {
        "system": "http://terminology.hl7.org/CodeSystem/v3-ActCode",
        "code": "OPTIN"
      }
    ]
  },
  "provision": {
   "actor": [
      {
        "role":{
          "coding": [
            {
              "system": "http://terminology.hl7.org/CodeSystem/v3-ParticipationType",
              "code": "PRCP"
            }
          ]
        },
        "reference": {
          "identifier": {
            "system": "urn:oid:2.16.840.1.113883.2.4.6.1",
            "value": "00000007"
          }
        }
      }],
    "period": {
      "start": "2016-06-23T17:02:33+10:00",
```

(continues on next page)

(continued from previous page)

```
      "end": "2016-06-23T17:32:33+10:00"
    },
    "provision": [
      {
        "type": "permit",
        "action": [
          {
            "coding": [
              {
                "system": "http://terminology.hl7.org/CodeSystem/consentaction",
                "code": "access"
              }
            ]
          }
        ],
        "class": [
          {
            "system": "http://hl7.org/fhir/resource-types",
            "code": "Observation"
          }
        ]
      }
    ]
  }
}
```

2.5 Supported consent flows

2.5.1 Create Genesis Consent State

Enforcing unique constraints for distributed is extremely hard (if not impossible). The only tool we have is that we can enforce a state has to be consumed (just once) and this constraint is enforced by a notary. Using this knowledge, a genesis state has to be created for a *DPC* record before consent is added. The creation of the genesis state is node local, which makes it possible to apply a unique constraint through a normal database transaction. The creation of the genesis record is done by the corda bridge.

2.5.2 Create Consent Branch

This flow is initiated when a *DPC* has to be changed. The flow creates a *ConsentBranch* state without any validation proofs of any party. All nodes in the Nuts network must listen to these state changes and start the approval process in *Service Space*. This will validate if the record is correct and if the patient is indeed a patient for the given Custodian. An attachment must be uploaded to the initiating node or the transaction will fail. Each party checks if the given proofs are valid and if the used public key is indeed the one used by that party (via Nuts Registry).

The *branch* and *merge* concepts are taken from Git. The idea is that when additional consent is added a merge should not be a problem (like new files in Git). But if an existing record is changed, this can only be done if there are no conflicts. The notary will define in which order merges are done in the case of a race-condition. Using this approach, adding multiple parallel records at once shouldn't be a problem.

2.5.3 Sign Consent Branch

Whenever a *Consent Branch* is created, other parties have to approve the new request. This is done via the *Sign Consent Branch* flow. The flow adds the signatures of a care provider to the *ConsentBranch* state and starts a transaction. The transaction will then be validated by all nodes. Each care provider has to approve a *ConsentBranch* before it can be merged into a *DPC* record. The signatures are generated by the private key of the care provider, so any node can validate it in *service space*.

2.5.4 Merge Branch

The *Merge Branch* flow takes a *ConsentState* and a *ConsentBranch* and merges it into a new *ConsentState*. It can only merge these two if the *ConsentBranch* only added new consent or if no other *Merge Branch* flow has been performed since the *Create Consent Branch* flow.

2.6 Identifiers

The main Nuts components rely heavily on identifiers. In most of the source code these are referred to as `Identifier` and are coded as strings. This makes the code reusable for all kind of use-cases. Within the Nuts *spaces* identifiers are just used to link different records. On the boundary of *service space* and *vendor space*, however, some choices have to be made in order for vendors to understand the different identifiers and connect them to their own logic.

Nuts identifiers use **URN** style representation and use existing URN's where available. An identifier consists of two parts, the URN and the unique representation within the URN namespace. See https://en.wikipedia.org/wiki/Uniform_Resource_Name or https://tools.ietf.org/html/rfc8141 for how URN's are constructed.

The complete Nuts identifiers thus consists of the URN followed by a colon and then the identifying value.

Identifier	Known as	Used in	Description
urn:oid:2.16.840.1.113883.2.4.6.3	BSN	Consent	'Burgerservicenummer' used within the Netherlands for identifying a patient.
urn:oid:2.16.840.1.113883.2.4.6.1	AGB-Code	Consent/Registry/Auth	The AGBCode as maintained by Vektis for both Organizations and professionals.
urn:oid:1.3.6.1.4.1.54851	Nuts root OID		Root OID used for mounting point of Nuts classifiers.
urn:oid:1.3.6.1.4.1.54851.1	Nuts consent classes	Consent	Consent classifier ID, eg: urn:oid:1.3.6.1.4.1.54851.1:MEDICAL.
urn:oid:1.3.6.1.4.1.54851.2	Nuts endpoint types	Registry	The type of endpoint a URL points to, eg: Consent, Registry or FHIR.
urn:oid:1.3.6.1.4.1.54851.3	Nuts domain	Registry	Domain supported by Nuts in which a vendor/organization operates. Supported values: "healthcare", "social", "pgo" or "insurance".
urn:oid:1.3.6.1.4.1.54851.4	Nuts vendor ID	Registry	Identifies a registered vendor. Contains the Chamber of Commerce registration number, encoded as a string.
urn:ietf:rfc:1779	X500Name	Registry	Name notation used in X509 Certificates. Identifies a Consent Corda node.
urn:ietf:rfc:3986	URI	Consent/Registry/Auth	Indication the value is of type URI, should be used to indicate an OID

The choice (for now) has been made to use URN and OID style identifiers. URL style identifiers commonly used as namespaces seem to be all over the place and each new project or initiative declares its own namespace for the same identifier. We'd like the identifiers to be more static.

2.6.1 Examples

BSN

```
urn:oid:2.16.840.1.113883.2.4.6.3:999999990
```

AGBCode

```
urn:oid:2.16.840.1.113883.2.4.6.1:00000007
```

Hospital OID

In this case the hospital has its own OID.

```
urn:ietf:rfc:3986:2.16.840.1.113883.2.4.3.55
```

X500Name

```
urn:ietf:rfc:1779:O=Nuts, OU=Healthcare, C=NL, ST=Gelderland, L=Eibergen, CN=nuts_
↪corda_development_local
```

Consent Endpoint

Endpoint to connect with to synchronise consents

```
urn:oid:1.3.6.1.4.1.54851.2:consent
```

2.7 Login Contracts

In the Nuts network, the identity of a user is universally valid. This means the whole network trusts authorities like the BIG, CIBG and BRP instead of the unknown sysadmin and processes of a random care organisation.

When a user wants to access data stored in a different EHR system than they are currently using, they must give their own system consent to request data.

So, we have several parties:

- A user, probably a care professional or patient (User)
- A trusted registry which administers and issues attributes about the user such as an identity or medical qualification (Issuer)
- The users own software system (Acting Party)
- Another system containing data the user wants to access. (Verifier)

The consent must:

- be human readable
- be machine readable
- be final in time
- only be usable by the users software provider
- temper proof
- versionable
- signable with attributes
- support multiple languages
- should fit in the http AUTHENTICATION header
- **contain the following attributes**
  - The identity of the user
  - The software system given consent to
  - The scope of the consent
  - The purpose of the consent
  - Period when the consent is valid

2.7.1 Example

Using regular expressions, a machine can extract all the relevant variables:

```
/(language):(Contract-name):(version) Ondergetekende geeft hierbij toestemming aan
(Acting Party) om namens (Legal Entity) en ondergetekende (Scope) ten behoeven van␣
↪(Purpose).
Deze toestemming is geldig (period)./
```

Examples:

NL:Login:v1 Ondergetekende geeft hierbij toestemming aan *SoftwareApplicatie* om uit naam van *Zorgorganisatie* en ondergetekende *data op te vragen, data aan te passen* ten behoeven van *het verlenen van goede zorg*. Deze toestemming is geldig tot *2019-02-06T21:00*.

NL:Gegevensverwerking:v3 Ondergetekende geeft hierbij toestemming aan *Medians* om uit naam van ondergetekende *data op te vragen, data te verzamelen* ten behoeven van *het doen van medisch onderzoek*. Deze toestemming is geldig tot *2020-02-06T21:00*.

NL:Inzage:v2 Ondergetekende geeft hierbij toestemming aan *Alle cardiologen* om *data op te vragen* ten behoeven van het *verlenen van goede zorg*. Deze toestemming is geldig van 2018-01-01T00:00 tot 2030-12-31T23:59

You can try this simple example over here: https://regex101.com/r/sHfsKq/1

2.7.2 Matching groups

Language

We can use human readable Language tags.

Contract-name

We can have multiple contracts like login, giving consent for treatment, authorization etc. The contract name is localized.

Version

We should be able to update contracts as they might change over time. Since the version indicator will be displayed in the contract, it is important to choose a form which does distract or confuse the user. Think of a scheme like v1, v2 etc.

Acting Party

Purpose: Only the acting party is given the permission.

In case of a Nuts party, We can use the X.509 CommonName used in the Directory service for software names. In case of people, we can use relevant attributes such as a name, medical expertise or medical registration number (BIG, AGB).

Legal Entity

Purpose: limit the consent to only a single care organisation.

The name of the care organisation must match with on in the Nuts registry.

Period

Purpose: limiting the time frame a contract is valid.

We should agree on a human and machine readable time format like ISO 8601: https://xkcd.com/1179/

Consent scope

Purpose: limit the possible operations the acting party can perform on the data.

Actions like: collect, access, use, disclose or correct: https://www.hl7.org/fhir/valueset-consent-action.html

Purpose

What will be the reason for performing the operations on the information? We can encode our *Purpose* field based on the FHIR consent scope set: https://www.hl7.org/fhir/v3/PurposeOfUse/vs.html

2.7.3 Signing with attributes

Nuts is made for many different user groups: GPs, patients, guardians, informal care givers etc. All these people have different relevant properties (attributes). Therefor we need an universal authentication system which can handle many different attributes. https://privacypatterns.org/patterns/attribute-based-credentials

Attribute Based Credentials (ABC)

In order to sign a contract we need an application which support these attributes. Currently in the Netherlands there is one system which gains in popularity: IRMA (I reveal my attributes) from the privacy by design foundation: https://privacybydesign.foundation/.

Using ABC decouples the issuer from the verifier. This improves the privacy of the user, eliminates the need for 1-to-1 API implementations and the need for complicated legal documents between the parties since the users decides what they will do with their own data.

Note: It is important to note that the workings of Nuts are not limited to IRMA.

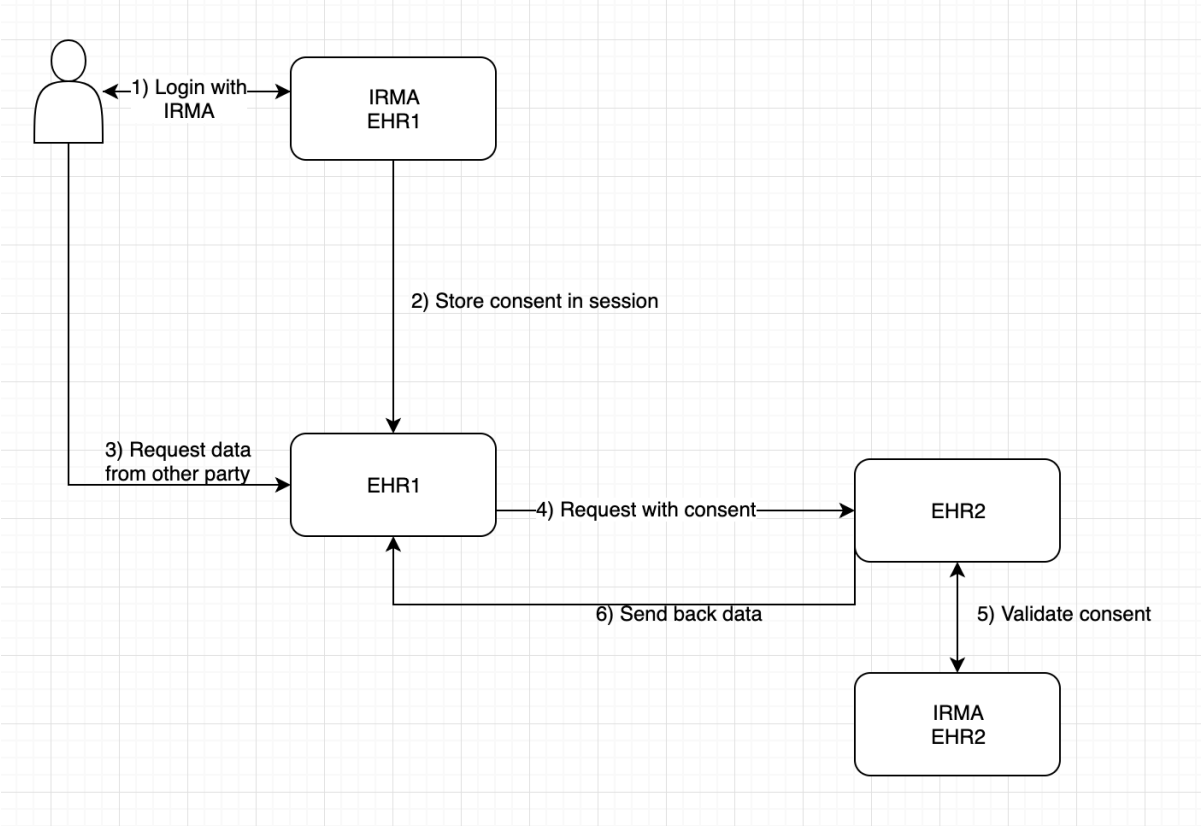
Every node in the Nuts network sets up its own trusted IRMA server with the Dutch IRMA key chain so it can independently validate claims and contracts.

2.8 Nuts Network engine

This section describes the workings of the Nuts Network engine.

2.8.1 Status

This engine intents to provide a decentralized, persistent ledger for Nuts nodes to share information. While it is currently Work-in-Progress, we intent to migrate the Registry Github-sync mechanism and Corda consents to it. These are known as *applications* on the network. The Registry application will be realized first and will serve as Proof-of-Concept (POC) for the network as a viable distributed ledger for Nuts. If successful we can decide whether to migrate Corda consents to the network too.



The Registry application will first operate alongside the Registry's Github-sync as to assess learn how the networks behaves in real-world scenarios and to assess the POC. If successful we can decide whether switch from Github-sync to the network application.

2.8.2 Design

The business end of the engine consists of 5 layers:

Applications

When services want to exchange information with other Nuts nodes they can build their application on top of the *DocumentLog* layer. Internal (to the network engine) applications can feed back information into one of the lower layers: the *NodeList* application is used to discover new potential peers for the P2P layer.

Interfaces

The Network engine has 2 interfaces:

1. REST interface for diagnostics (like all engines) and for applications to query/add documents.
2. Message queue for applications to subscribe on document events.

Applications subscribe to specific document types (based on the **type** property). Documents that are needed for the basic functionality of the network use the **nuts.** prefix:

- **nuts.node-info**: documents produced/consumed by the NodeList

DocumentLog

The DocumentLog provides a persistent document log which contains all documents on the network (known by the local node), fed by the network itself (remote peers) and the local node. Through this layer applications can add and query documents. It makes sure the documents are in a consistent order and tries to build a complete view of documents (and their contents) on the network.

While every participant receives every document hash, access to some documents might be restricted to a limited set of parties. In this case, the document contents will only be present in the DocumentLog of those parties.

Protocol

The protocol layer is responsible for knowing how to query the network and how to respond to queries from peers.

P2P

The P2P layer is responsible building and maintaining network connections to our peers, authenticating them and to provide a channel for the protocol to communicate with them (our peers).

2.8.3 Components

The following diagram shows which components make up the engine and how they relate.

2.8.4 Configuration

The following diagram shows which configuration artifacts are used by the engine and where they come from.

2.9 Nuts Auth JWT specification

The JWT is specified as:

```
{
    "iss": "urn:oid:2.16.840.1.113883.2.4.6.1",
    "sig": base64 encoded signature
    "type": type of the encoded signature
}
```

2.9.1 Iss

The issuer contains the urn of the requestor (actor). The token has been signed with the private key of the requestor. This is to make sure that a token cannot be reused be any other party to make requests in the Nuts network.

Note: Since the nuts token is signed with the private key of the requester, it is not trivial to verify the signature of the token. When recieving a request, any token signature verification steps must be postponed until it is clear a token is not a nuts token.

2.9.2 Sig

The signature contains a base64 encoded string. The *type* attribute determines the content of this signature.

2.9.3 Custodian selection

A single API endpoint can provide access to the data of multiple care providers (e.g. a multi-tanant SaaS). Therefor, when requesting data, an actor needs to provide the custodian of the data it wants to request. For now, an actor must provide the **X-Nuts-Custodian** header with the urn based value in the request.

2.10 Nuts Auth JWT specification (Legacy)

Note: this specification is for the JWT used in version <= 0.13.

The JWT is specified as:

```
{
    "iss": "nuts",
    "sub": "urn:oid:2.16.840.1.113883.2.4.6.1",
    "nuts_signature": {...irma based signature...}
}
```

2.10.1 Iss

The issuer in the JWT is always *nuts*. This allows the API of the care provider to switch between validation mechanisms: When the Issuer is *nuts* post the token to the token check endpoint, if not, do your usual checks.

Note: Since the nuts token is signed with the private key of the requester, it is not trivial to verify the signature of the token. When recieving a request, any token signature verification steps must be postponed until it is clear a token is not a nuts token.

2.10.2 Sub

The subject contains the urn of the requestor (actor). The token has been signed with the private key of the requestor. This is to make sure that a token cannot be reused be any other party to make requests in the Nuts network.

2.10.3 Custodian selection

A single API endpoint can provide access to the data of multiple care providers (e.g. a multi-tanant SaaS). Therefor, when requesting data, an actor needs to provide the custodian of the data it wants to request. For now, an actor must provide the **X-Nuts-Custodian** header with the urn based value in the request.

2.11 Nuts Registry Data Model

The Nuts Registry is event-sourced, meaning its data is stored as events which are replayed to determine the current state. The event store is append-only which makes it easier to migrate to a distributed architecture.

An event contains the following fields:

Field	Type	Since	Description	Example
issuedAt	string	v0	Time at which the event was issued in UTC	1970-01-01T00:00:00Z
type	string	v0	Type of the payload	RegisterVendorEvent
jws	string	v0	Optional. JWS-encoded signature of the event	
payload	object	v0	Actual payload of the event	`{"Message": "Hello, World!"}`
version	int	v1	Version of the event	1
ref	string	v1	Hex-encoded reference to this event	67f732c4a421c8d7e097dfa55a27b67b4c5fbd9e
prev	string	v1	Hex-encoded reference to the previous event	67f732c4a421c8d7e097dfa55a27b67b4c5fbd9e

It contains 2 sections, its headers (`issuedAt`, `type`, *jws*, `version`, `ref`, `prev`) and a human-readable copy of the content of event (`payload`). We call it a copy because the authenticated content of the event is found encoded inside the *jws* field. The `payload` field is there for ease of use but will be removed in a future version and thus should not be relied on. The `type` field defines the type of the event (e.g. `RegisterVendorEvent`), indicating how to interpret the payload.

2.11.1 Signing

The events are signed to assure authenticity (proof that the event was actually created by the expected entity) and integrity (proof that the contents weren't altered by an attacker). The entity that 'owns' the data the event generates must be the one signing the event (see the table below).

Signature format

The signatures are in the JWS (JSON Web Signature) format. Since every key should be associated to a known entity, the JWS will contain an X.509 certificate (in the `x5c` field) describing the entity owning the key. The algorithm used for constructing the JWS is RS256 (RSA with SHA-256 hashing function).

The JWS also contains the actual payload of the event.

Validation

To validate an event signature, the following checks must be performed:

1. Is the JWS parsable?
2. Does the JWS contain an X.509 certificate chain (in the `x5c` field)?
3. Is the certificate meant for signing (key usage must contain `digitalSignature`)
4. Is the certificate (extracted from the chain) trusted?
5. Was the certificate valid at the time of signing (`issuedAt`)?

6. Is the owning entity of the certificate (e.g. a vendor or organization) the one we expected to sign the certificate (see *Owner check* in the table below)?
7. Is the JWS signed using the RS256 algorithm?
8. Is the used RSA key of sufficient length (>=2048 bits)?
9. Is the JWS signed with the private key belonging to the public key in the certificate?

When an event payload containing a CA certificate is successfully validated, it should be added to the node's trust store so that future events which are signed using certificates issued by the (CA) certificate can be validated.

Event	Signer	Owner check
RegisterVendor	Vendor	`Event.Payload.Vendor == Certificate.SubjectAltName[Vendor]`
VendorClaim	Vendor	`Event.Payload.Vendor == Certificate.SubjectAltName[Vendor]`
RegisterEndpoint	Organization	`Event.Payload.Organization == Certificate.SubjectAltName[Organization]`

Note: The Nuts Foundation will act as Root Certificate Authority so that intermediates are issued by an entity which is trusted by all participating parties. However, this Root Certificate Authority isn't operational at the time of writing so vendors are expected to self-sign their own CA certificates in the meantime. This means when validating a `RegisterVendor` event the certificate which signed the JWS will be self-signed and thus can't be validated. **This is the only case** where an unvalidated certificate should be added to the trust store.

2.11.2 Calculating event `ref`

To calculate an event's `ref` follow these steps:

1. Take all fields from the event applicable for the event's version (see the data structure table described earlier) but leave out `ref`.
2. Marshal to canonicalized JSON using Rundgren JSON Canonicalization Scheme (draft v17).
3. Hash the canonicalized JSON using SHA-1.

When representing `ref` in JSON it should be lowercase, hex-encoded.

2.11.3 Versioning

Events have a `version` field indicating the version of the data structure. New versions might introduce new fields or change the datatype of existing fields (although this change must be backwards compatible.

Version	Change
0	Any version before introduction of `version`
1	`version`, `ref` and `prev` added
2 (planned)	JWS signed payload is canonicalized before hashing

2.12 Service space event specification

Below is the flow diagram of how a consent request is transformed to a distributed-privacy-consent record. Blocks represent actions or commands and the hexagons represent events. The event model and the different events are descirbed further down.

2.12.1 Event model

```
event:
    UUID: string                    # V4 UUID
    name: string                    # event name, see table
    retryCount: int                 # 0 to X
    externalId: string              # ID calculated by crypto using BSN and private␣
↪key of custodian
    consentId: string               # V4 UUID assigned by Corda to a record, either a␣
↪ConsentBranch or ConsentState
    initiatorLegalEntity: string    # urn style identifier of the initiating␣
↪legalEntity, used to select the party who's finalizing the request
    transactionId: string           # V4 UUID identifying a possible Corda transaction␣
↪that was started by this event chain
    payload: string                 # Base64 encoded NewConsentRequestState JSON as␣
↪accepted by consent-bridge (:ref:`nuts-consent-bridge-api`)
    error: string                   # error reason in case of a functional error
```

Payload per event

Event name	Payload	Description
ConsentRequest constructed	FullConsentRequest	As defined by consent bridge, the attachment signatures will be an empty list
ConsentRequest in flight		
ConsentRequest flow closed		
ConsentRequest flow errored		
Distributed ConsentRequest received	FullConsentRequest	In principle the same as the NewConsentRequest above, but it'll contain AttachmentSignatures when available This event will also be the starting point for any other node than the initiating one
All signatures present		
ConsentRequest in flight for final state		
ConsentRequest valid		
ConsentRequest acked		
ConsentRequest nacked		
Attachment signed	AttachmentSignature	A single signature will be present in the event. When processed by Corda, the NewConsentRequest will be back with this signature included
ConsentRequest flow errored		
Consent distributed	ConsentState	The final consent state

2.12.2 Event types

ConsentRequest constructed

A new consent request has been POSTed to the *consent-logic* module. It is checked, an externalId is added and it is converted to a FHIR model. The request is encrypted with the pub keys of all recipients and the published as an event with state `consentRequest constructed`.

ConsentRequest in flight

`consentRequest constructed` Events are handled by the *corda-bridge*. The *corda-bridge* find the nodes to be involved and submits a transaction to Corda. The transactionId from Corda is added to a new event with state `ConsentRequest in flight` It'll then start polling for the initiated transaction to give feedback about its state. If all nodes have signed (including the notary), the *corda-bridge* publishes the event with added consentId and state: `Distributed ConsentRequest received`.

ConsentRequest flow closed

Event that is published when a Corda close flow has been executed. This is the case when another node nacks the request. The error field will give information about the closure reason.

ConsentRequest flow errored

Event that is published when a Corda flow was closed with an error reason. The error field will give information about why it failed.

ConsentRequest flow success

Event that is usefull to debug the current state of the event flow. Will be visible in the consent store. It'll not be input for further processing, since that will be initiated by Corda events.

Distributed ConsentRequest received

The *corda-bridge* receives events from Corda when transactions are completed. It'll find the corresponding event with state: `ConsentRequest in flight` or `ConsentRequest flow success` or when another node initiated the transaction, it'll create a new event from scratch. Either way a new event with state: `Distributed ConsentRequest received` is created.

All signature present

`Distributed ConsentRequest received` events are processed by the logic module. If all signatures are present, it'll generate an event with state `All signatures present`.

ConsentRequest in flight for final state

When a consent request is nacked or when the initiator has concluded all signatures are present, the correct flows are called by the bridge and an event is published: `ConsentRequest in flight for final state`. This indicates that no further logical processing is needed.

ConsentRequest valid

`Distributed ConsentRequest received` events are processed by the logic module. If not all signatures are present, it'll validate the record and check if all current signatures belong to the involved parties. When ok, a `ConsentRequest valid` event is published. This event is picked up by the logic module and auto-acked (for example when this node == the initiator) or the event must be picked up by *vendor space* for manual acking.

Note: can Corda do this check in the contract using an Oracle in the form of the registry? On Github

---

ConsentRequest acked

Either the logic module or from *vendor space* an `ConsentRequest acked` event is produced indicating that the subject is indeed a patient in care by the given legalIdentity.

Attachment signed

`ConsentRequest acked` events are picked up by the logic module and a signature is produced. This will result in a `Attachment signed` event. This event is picked up by the bridge which will initiate an AcceptConsentRequest flow. This will result in an `ConsentRequest in flight` event. From here-on the event flow tree is reused.

Consent distributed

After `ConsentRequest in flight for final state` Corda will transform the `ConsentRequestState` to a `ConsentState`. This event is picked up by the bridge to publish a `Consent distributed` event.

Completed

From the `Consent distributed` event, consent records are persisted in the *consent-store*. The event chain is completed and will enter the `completed` state.

Error

If for some reason, an event enters the error state, the error field of the event will show the explanation. Since the event log is a circular log, errored events will not survive restarts if they are older than X (depending on the log size). It is recommended to store errored events by parsing the regular error logs and storing them somewhere. An error event published to the error channel will not be propagated across nodes, an error event published to the regular channel will be picked up an synchronized across nodes.

2.12.3 Channels and queues

Most messaging/queueing technologies share the notion of the separation of channel and queues. Message are published to channels and stored in queues. All queues are durable which means they will survive a restart/crash.

Channel | Queue | Consumer | Description |

+======================+========================+================+==============================

| consentRequest | consentRequest | eventStore | The event store processes all events and stores the current state in a db | | +——————+————-+————————————————————————+ |

| consentRequest | consentLogic | The validation module only processes new events and checks if they are correct | | +——————+————-+————————————————————————+ | | consentRequest | consentBridge | The bridge listens to events that are ready to send to Corda | | +——————+————-+————————————————————————+ | | consentRequest | consentStore | The consent store handles events that are finalized and can be stored in a persistent data store | +————-+——————+————-+————————————————————————+ | consentRequestRetry | consentRequestRetry | eventOctopus | General retry queue where events to be retried are sorted | +————-+——————+————-+————————————————————————+ | consentRequestRetry-X | consentRequestRetry-X | eventOctopus | Where X is the retryCount.

Events are picked up and the service sleeps untill the event can be | | | | | re-published to the consentRequest channel | +————————+—————————+——————-+———————————————————————————————+

2.12.4 Retry mechanism

Some errors may be caused by timeouts or poorly working infrastructure. To remedy this, events can be retried. Events that should be retried must be published to the *consentRequestRetry* channel. The *eventOctopus* will sort the event to a different queue based on the *retryCount* (or save it as an error if the max count has been reached). Events that are picked up by the different retry queues will remain there until the timeout has been reached (by means of delay acking the message). The different queues have a different waiting time till the events are republished to the main channel. This can be configured by the *maxRetryCount* and *incrementalBackoff* config variables. The *incrementalBackoff* multiplies the waiting time of the previous queue. The default settings of 5 retries and an incremental backoff of 8 means that the waiting times for the different queues are: 1s, 8s, 64s, 512s, 4096s or 1s, 8s, ~1m, ~8m, ~1:08h.

2.12.5 Implementation

Nats is used as messaging system with Nats-streaming as event log. The event store will be implemented with an in-memory SQLite DB. The *Nats* service is part of the *nuts-event-octopus* and is embedded within the `nuts` service executable.

Some introduction to the general architecture

CHAPTER 3

RFCs

3.1 RFC: Access Tokens

Access tokens are short lived opaque tokens which accommodate an API request and refer to contextual information of the request. Such an API request is scoped by the token to an actor, subject and custodian. Optionally a reference to a specific consent record can be provided.

3.1.1 Status of this RFC

This RFC is a work in progress. See the *TODO* section below. To provide comments create an issue on the Github repository containing this documentation.

3.1.2 Motivation

Originally the plan was to only provide the Actor's *Login Contracts* along the API request, assuming this was sufficient. However we encountered situations where more information was needed.

Size

The more attributes are requested to sign the IMRA login contract, the bigger the resulting JSON hash gets. So big that, when base64 encoded and signed, it exceeds the often used http header max size of 8kb. Although header sizes are not officially limited, approaching this limit will likely result in unforeseen problems.

Subject scoping

How does the custodian's server know which subject the data request is about? Some protocols contain a patient id in the URL, others require inspecting the body or headers. In order for Nuts to have a minimal implementation effort, it is important to refer to the patient in a uniform way.

Custodian scoping

Some protocols run on a multi tenant environment which need a way of specifying the custodian the request is about. Just like the subject, in order for Nuts to have a minimal implementation effort, it is important to refer to the custodian in a uniform way.

Cost of checking consent and Identity

When performing multiple request for a triple of custodian|subject|actor, the *Login Contracts*, and consent must be verified each request. The crypto behind IRMA takes some time, adding up to the response time of these requests.

3.1.3 Limitations

Because it is necessary to request a new access token for every combination of subject custodian and actor, bulk operations, like getting reports for many patients at once, is not possible. We could consider allowing list of subjects during the token requests.

3.1.4 Mechanics

To resolve above issues, we'll introduce *access tokens*. Access tokens refer to a context of the current request. This information can be embedded in an encrypted access token or can be stored on the authorization server. This information contains of the subject, actor and custodian. Additional information can be stored like, token expiration and consent reference.

To **obtain the users identity**, the users follows a standard Login flow presenting its attributes using IRMA, signing the LoginContract. The signed contract can be stored by the users XIS in a session for the validity of the contract.

Once the user wants to request data about a subject, a **access token must be obtained**. The access token can be acquired using an OAuth 2.0 flow with the *urn:ietf:params:oauth:grant-type:jwt-bearer* grant_type. This profile is described in RFC7523.

For Actor identification the requests will be performed over a mutual SSL connection. The client will use a certificate signed by the CA of the custodian. This will be described in another RFC.

As the rfc7523 describes, a JWT Bearer token should be build and posted to the Authorization Server (Nuts node or own implementation). The Authorization Server checks all requirements like a valid consent, validity of the access token, match of clientID and Actor etc. If everything is valid, it creates and returns the access token.

The client can now perform API requests using the same mutual SSL connection providing the access token along with the request. For example, in the case of a HTTP REST call in the form of a bearer token in the Authorization Header. Contrary to the original OAuth 2 definition of a Bearer token, the token is bound to the client by its two way (mutual) SSL certificate.

The XIS receiving the access token can retrieve the context by token introspection.

3.1.5 The endpoint

A vendor can implement this flow using its own existing infrastructure or use a by Nuts provided minimalistic implementation.

The address of the Authorization Server endpoint must be provided in the resource server's registry entry in the properties object under the key *authorizationServerURL*.

A complete registry entry for an sso endpoint can look like this:

```
{
    "URL": "http://sso.custodian.local/land",
    "endpointType": "urn:oid:1.3.6.1.4.1.54851.1:nuts-sso",
    "identifier": "7b8f7852-d218-4242-8406-39cf6abcde58",
    "properties": {
        "authorizationServerURL": "http://nuts.custodian.local
    },
    "status": "active"
}
```

3.1.6 The JWT

The JWT used to obtain the token should consists of the following fields:

```
{
  "iss": "urn:oid:2.16.840.1.113883.2.4.6.1:48000000",
  "sub": "urn:oid:2.16.840.1.113883.2.4.6.1:12481248",
  "sid": "urn:oid:2.16.840.1.113883.2.4.6.3:9999990",
  "aud": "https://target_token_endpoint",
  "usi": {...Base64 encoded IRMA based signature...},
  "osi": {...hardware token sig...},
  "con": {...additional context...},
  "exp": max(time_from_irma_sign, some_limited_time),
  "iat": 1578910481,
  "jti": {unique-identifier}
}
```

iss

The issuer in the JWT is always the actor, thus the care organization doing the request. This is used to find the public key of the issuer from the Nuts registry.

Note: Since the nuts token is signed with the private key of the requester, it is not trivial to verify the signature of the token. When receiving a request, any token signature verification steps must be postponed until it is clear a token is not a nuts token.

---

sub

The subject (not a Nuts subject) contains the urn of the custodian. The custodian information is used to find the relevant consent (together with actor and subject).

sid

The Nuts subject id, patient identifier in the form of an oid encoded BSN.

aud

As per rfc7523, the aud must be the token endpoint. This can be taken from the Nuts registry.

usi

User signature. This is the IRMA signature presented to the user. Base64 encoded. It contains the users identity and consent for the vendor to use the Nuts network on its behalf.

osi

Ops signature, optional signature coming from a hardware token, indicating the user belongs to the issuer organization. Can be linked to the Nuts registry. This mechanism is used to establish an employer relationship without actual placing personal information into the registry.

con

Base64 encoded JSON representing key-value pairs for additional context for the requested access token. Such as task flow selection.

exp

Expiration, should be set relatively short since this call is only used to get an access token. Must not be bigger than the validity of the Nuts signature validity.

iat

Issued at. NumericDate value of the time at which the JWT was issued.

jti

Unique identifier, secure random number to prevent replay attacks. The authorization server must check this!

3.1.7 TODO

Some things have to be defined:

- the exact formats of API calls
- the mechanisms of mutual SSL

3.2 RFC: Distributed Registry

Note: This document is going to be replaced by Nuts Specification RFC003, RFC004 and RFC005.

The goal of this system is to provide a registry in which parties can:

- Lookup endpoints which can be used to exchange data.
- Search for care organisations connected to the Nuts network (through their vendor).

3.2.1 Non-functional Requirements

Availability

- Every consumer of the registry should have its own instance, because:
  - End users will be using the registry to search for care organisations, probably using full-text search. Having a local copy gives the best performance and user experience.
  - Remote systems and networks can be unreliable, having a local copy ensures availability.
- Every instance should connect to other instances and be open for connections from other instances, which provides better availability than relying on a centralized instance.

Durability

- Every instance should maintain a complete copy of the registry (including the complete history), and be willing to share it with other instances. This helps to mitigate the risk of data loss.

Security

- Data in the registry should be digitally signed so that the identity of the producing party and the integrity of the data can be reassured.
- Connections (between instances) are secured using 2-way SSL. Vendor instance identify and authenticate themselves to their peers using their node identity certificate.

Maintainability

- Changes should be backwards compatible so that new features don't break historical data.
- New instances must be able to get the full history after joining.

3.2.2 Roles

Role / Entity	Description
Care organisation	Healthcare organisation (e.g. hospital, insurance company, who wishes to exchange data with other care organisations through their respective vendors using Nuts.
Care organisation signatory	Person within a care organisation who is authorised to sign in the name of their organisation.
Vendor	Organisation who supplies and operates a care organisation's software systems containing medical data.
Nuts administrator	Person within the Nuts Foundation who is authorised to accept registrations from vendors.
Node	Vendor's system used to communicate with the network, conform the Nuts specification.

3.2.3 Architecture

The data contained in the registry is modelled as a chain of immutable events, where every event refers to its preceding event. Since events are immutable, every change (or deletion) is recorded as a new event. Each event is signed so that every participant can validate the integrity of the event chain.

Every event contains the following fields, aside from its event-specific data:

Field	Format	Description
ID	Hash	Uniquely identifies this event
Previous event	Hash	Refers to the previous event in this stream
Date of issue	Timestamp	Instant the event was emitted
Signature	RFC 7515 JWS	Signature which authenticates the event
Payload	JSON	Payload of the event.

3.2.4 Processes

Registering a vendor

This process makes a vendor known to the network. Afterwards:

- The vendor is known to the network.
- The vendor can claim their relationships with care organisations.

Limitations:

- For now, we only support 1 node per vendor.
- For now, we only support 1 vendor per care organisation (but a vendor can serve multiple care organisations).

Steps:

1. Node registration:
    1. The vendor generates a CSR (for the node's CA certificate) and submits it to Nuts Discovery.
    2. A Nuts administrator accepts the CSR and issues the node CA certificate.
    3. The vendor issues node identity certificate (for its node) using its CA.
    4. The vendor submits its node information (node identity certificate, node address, ...) to Nuts Discovery.
    5. The node information is published so that every (other) node in the network knows about the vendor's node.
2. Vendor registration:
    1. The vendor generates a CSR (for the vendor's CA certificate) and submits it to Nuts Registry.
    2. A Nuts administrator accepts the CSR and issues the vendor CA certificate.
    3. The vendor issues its vendor certificate using its CA certificate. This certificate is used to sign events.
    4. The vendor creates the registration event and signs it using the vendor certificate.

Vendor CA certificate fields:

Field	Value
Validity	3 years
Key usage	Digital signature
CA?	true
Issuer	Nuts Vendor CA
Subject.CN	Vendor's name or unit within the organisation
Subject.O	Vendor's name
Subject.ST	Province in which the vendor is located
Subject.C	Vendor's country
Subject.serialNumber	Chamber of Commerce registration number of the vendor

Vendor Registration Event Payload

Field	Format
Vendor CA certificate	JWK encoded X.509

Note: We could add the vendor CA certificate to the X.509 trust chain when a certificate is referenced, making this event unnecessary. However, this bloats the event header so it's desirable to register it once using this event so it can be looked up every time a certificate (issued by this CA certificate) is validated.

Registering a vendor - care organisation relation

Through this process a vendor registers a care organisation as its client. Afterwards, the vendor can register endpoints for the care organisation it serves through its node.

Steps:

1. The vendor issues a certificate for the care organisation under its vendor CA certificate (see the table below for the prescribed fields of the certificate).
2. The vendor creates the claim event and signs it using the (just issued) .

Care organisation certificate fields:

Field	Value
Validity	3 years
Key usage	Digital signature
CA?	false
Issuer	Vendor CA certificate
Subject.CN	Care organisation's name or unit within the organisation
Subject.O	Care organisation's name
Subject.ST	Province in which the care organisation is located
Subject.C	Care organisation's country
Subject.serialNumber	Chamber of Commerce registration number of the care organisation

Vendor Claim Event Payload

Field	Format
Care organisation	URN
Vendor	URN
Start date	RFC 3339 timestamp
End date	RFC 3339 timestamp (optional)
Certificate	JWK encoded X.509

Note: In the ideal situation, the relationship is claimed 2-way: the vendor claims its care organisation as client, and the care organisation claims its vendor. This 2nd claim is authorised by a (digitally signed) registration of the Chamber of Commerce (Kamer van Koophandel) which identifies the care organisation. However, in the current state IRMA doesn't provide the Chamber of Commerce registration attribute (to identify organisation's authorised signatories). We expect this attribute to be available in the (near) future, so for now the care organisation's claim will be omitted. This will suffice for now, since every participating vendor is trusted since they're known by name and face by the Nuts Foundation.

Registering/updating an endpoint

Through this process a vendor can register (or update) an endpoint on which they use to serve data for one of their clients. Afterwards, other vendors can lookup the endpoint for data exchange.

Endpoint Registration Event Payload

Field	Format
Care organisation	URN
Vendor	URN
Endpoint ID	URN
Endpoint type	URN
Endpoint location	URL
Start date	RFC 3339 timestamp
End date	RFC 3339 timestamp (optional)

Requesting a vendor client certificate

This process is used by vendors to request a TLS certificate to authenticate themselves at other vendors' endpoints.

TODO

3.3 RFC: Medmij authentication flow using Irma

Warning: the proposal below is just that, a proposal. A prototype still has to be built.

The image below is a first attempt to solve the Medmij authentication flow using Irma instead of Digid. This is strictly a proposal and is being discussed with Medmij.

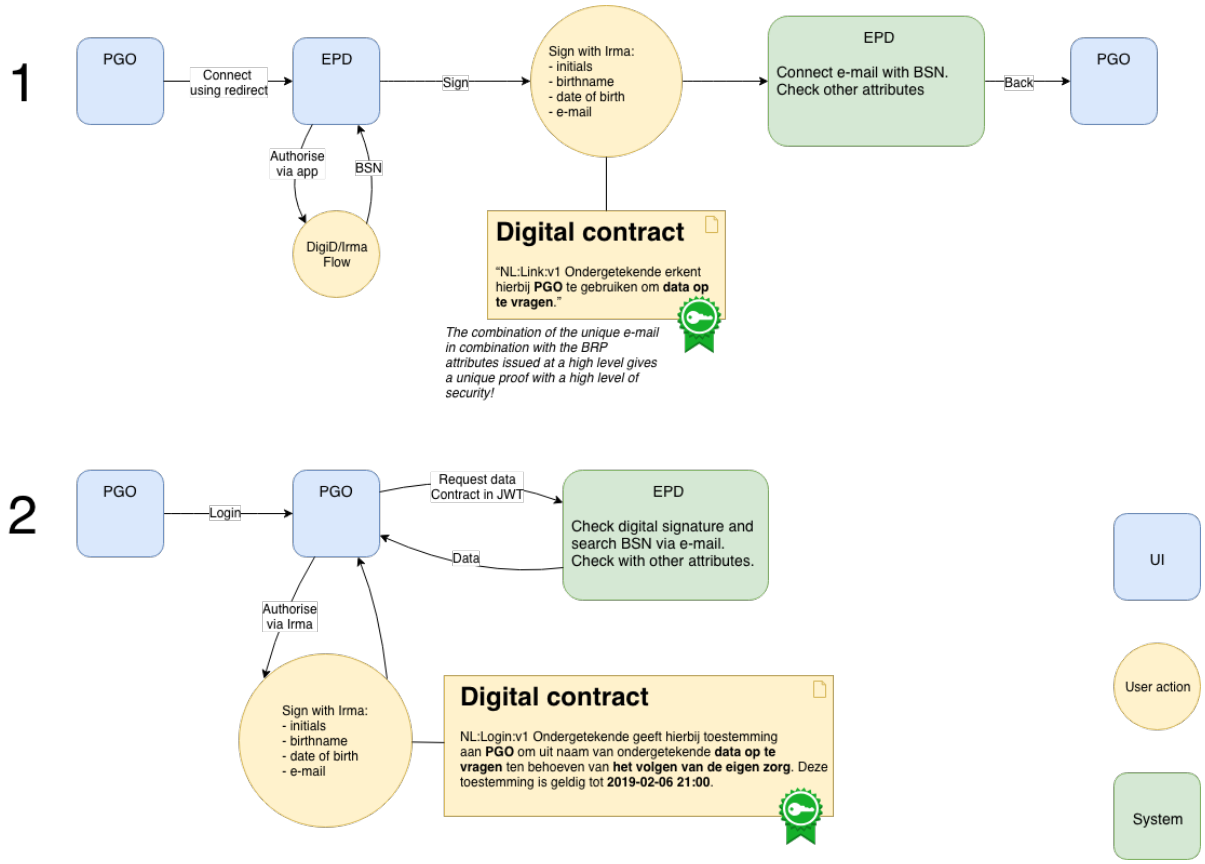


Fig. 1: Nuts Medmij auth flow

The flow above does two things: register the PGO user at a care organisation, this will link the BSN with a unique e-mail. Once registered, the second flow demonstrates how an Irma signature can be used to retrieve data from that care organisation. The fact that signatures are used is important because the information within the signature text is used to limit rights and to restrict the use of the signature between the PGO and the care organisation (linked to TLS connection). The main idea is that a unique attribute with a low security level combined with other attributes with a high security level creates a set that is both unique and secure.

3.4 1 RFC: Registry Updates

Note: This document is going to be replaced by Nuts Specification RFC003, RFC004 and RFC005.

3.4.1 1.1 Motivation

The Nuts Registry contains information about who (vendors, organizations) participates in the Nuts network and where to find the resources they're exposing through endpoint registration. Parties also use it to register their certificates so that other parties can verify the integrity and authenticity of exposed data.

The registry is modelled as an append-only event log which makes it easy to add new data (e.g. register an endpoint). However, there is currently no way to alter an data (e.g. rename an organization). This is an essential feature for any Nuts network which is managed by more than a single party, since altering data currently requires coordinated data pruning across all participating Nuts nodes.

This document proposes a mechanism for updating data in the registry.

3.4.2 1.2 Status of this RFC

DRAFT

3.4.3 1.3 Terminology

TODO

- Event: an event in the registry event log, e.g. the RegisterVendor event
- Event payload: the actual contents of the event, describing an object.
- Object: the thing an event describes; either a vendor, organization or endpoint.
- Event header: top-level properties of an event describing its payload.

3.4.4 1.4 Operations

The mutate operations that can be performed (by human operators or systems) on the registry are as follows:

Vendors

- Register vendor (currently supported)
- Update vendor name
- Reissue CA certificate
- Rotate keys (and reissue associated certificates)
- Deactivate/reactivate vendor

Organizations

- Register (claim) organization as vendor (currently supported)
- Update organization name
- Reissue CA certificate
- Rotate keys (and reissue associated certificates)
- Migrate organization to another vendor

Endpoints

- Register endpoint for organization (currently supported)
- Update endpoint (URL, type, properties)
- Deactivate/reactivate endpoint

Note: There are other operations that are somewhat related to the registry like issuing a TLS certificate to another vendor or revoking a certificate, but these don't mutate the registry and thus aren't relevant for this RFC.

3.4.5 1.5 Eventing style

From an operator's perspective (be it a human or system) they will be executing one of the operations described above. However, this isn't necessarily how it's they're mapped to events. These 2 styles have been taken into consideration, *domain driven* and *state driven*:

Domain Driven

Events are modeled in such a way that they resemble the domain (the operations described earlier) as close as possible e.g.: `RegisterVendor`, `UpdateVendorName`, `IssueVendorCertificate`, `UpdateOrganizationName`, `DeactivateVendor`, `ReactivateVendor`, `MigrateOrganization`, `UpdateEndpoint`, etc.

Events itself are very expressive and it is immediately clear what the event's intention. It makes it easy for Nuts nodes and auxiliary systems to rationalize or report about the data.

Determining the current state of an object is done by *replaying* all events, from the very first until the very last. When there are many events this can be an computationally expensive task. There's also the challenge of legacy: since every event is processed again and again long after it was published, every event needs to be supported indefinitely.

Note: Calculating objects' state by replaying cumulative events is known as *event sourcing*.

State Driven

The registry objects' (vendor, organization, endpoint) state are fully described by their respective "register"-events (`RegisterVendorEvent`, `VendorClaimEvent`, `RegisterEndpointEvent`). Since they describe the object's complete state, mutation could be done by simply replacing the entire state.

The data model of such a system is simple, since mutation is a matter of publishing the new state. Ceasing support for certain properties is simply a matter of ignoring them, making legacy data not much of a problem.

A downside is that a system needs to inspect differences between subsequent events ($E_n \ \Delta \ E_{n+1}$) to find out what changed and what operation was performed. Such system often implement a separate (audit) log for tracking changes.

3.4.6 1.6 Event sequencing

This section describes how events will be sequenced.

TODO

1.6.1 Challenges

Given the registry being an event driven distributed system, we see the following challenges:

1.6.1.1 Conflict detection and resolution

A lot can go wrong with an event driven distributed system, causing conflicting events which are distributed in the network, e.g.:

- a node publishes the same event twice (duplicate data),
- a node publishes 2 events with the same payload (redundant data),
- a node publishes 2 events with the same header but with a different payload (conflicting data).
- etc.

Because this threatens the integrity of the network, nodes must be able to detect and correct conflicting events.

1.6.1.2 Guaranteeing order

Since mutations should be applied in order

1.6.1.3 Actuality

Nodes use the information in the registry for communication with other nodes. Therefore it's important that nodes can verify the extend to which their view (on the distributed registry) is up-to-date. If a node assumes their view is up-to-date while it isn't, data exchange might fail. In worst case, data might accidentally be shared with another party (data leakage).

How do nodes know that their data set is valid and actual?

1.6.2 Solution

Conflict detection is relatively straightforward since every node should receive all data that is published to the network, so every node is able to perform checks to detect conflicts. For conflict resolution there are two styles taken into consideration:

1. Resolution which requires communication between nodes to decide which events to accept and which should be rejected. This often uses a form of voting to establish consensus.
2. Communicationless (as much as possible) resolution by structuring data in such a way that nodes can resolve errors on their own using a fixed set of rules.

Since communication (especially in a distributed system) adds many new potential fault modes a communicationless solution is preferred (option 2). This is generally known as a conflict-free resolution data type (CRDT). Since our events are basically an ordered append-only log, a set-based CRDT matches best. `G-Set` (Grow-only set)

1.6.3 Mechanics

Longest chain Garbage collecting tombstone set

Since events itself are immutable, only new events can be added to describe an object's new state. A subsequent event completely replaces the object's state described by the preceding event. The subsequent event must refer to the event it revises by its hash.

1.6.3.1 Event referral

An event refers to a preceding event (which it replaces) by its hash. There can only be one event referring to a particular event. In other words, subsequent events form a single stream without branching/merging:

```
+----+ replaces +----+ replaces +----+
|e(1)+<---------+e(2)+<---------+e(3)|
+--+-+          +----+          +--+-+
   ^                              ^
   |                              |
initial                         current
```

An event's hash is calculated by hashing its payload. The hash itself is specified the event's header in the `hash` field. If there's a preceding event its hash is specified in the `prev` field:

```
{
  "hash": "...",
  "prev": "...",
  "payload": {
    "identifier": "a09039da-f96b-4fd1-8925-fd28cb833159",
    "prop1": "Hello",
    "prop2": "World"
  }
}
```

Events should specify the `hash` header as an extra consistency check. When processing an event, its hash should be calculated and compared to the `hash` header. If it doesn't match the event must be rejected. For historic events which do not contain the `hash` header this check is skipped.

1.6.3.2 Hashing

An event is hashed by taking its payload and calculating a SHA-1 hash over its stringified payload. To maintain a consistent, deterministic hash across implementations and platforms the payload needs to be canonicalized before hashing. The Rundgren JSON Canonicalization Scheme (17) draft will be used for canonicalization since there's no industry standard or accepted RFC at the moment of writing. It also refers to compliant implementations, helping Nuts implementations for different platforms.

1.6.3.3 Integrity

To assure that updates are authentic (and not made by an attacker), ownership of the object must be verified before processing the event. This is done by asserting that the event was signed by the entity owning the object. In other words, vendors can only update their own organizations and organizations can only update their own endpoints, and so on.

Furthermore, to maintain functional integrity a subsequent event can only describe the same object as its preceding event. In other words given `event(1)` describing `vendor(1)`, `event(3)` referring to `event(1)` can only describe `vendor(1)`. This constraint is typically implemented by comparing object identifiers.

registry status (last update) op de status page

3.5 RFC: Single Sign On (SSO)

3.5.1 Motivation

Care professionals sometimes work with the same patients for several care organizations using different software vendors. Their daily work can greatly be improved when they can switch effortless between these applications without

providing new credentials on every switch. Therefor we propose a way of Single-Sign-On (SSO) for sharing a universal identity among different software applications, not necessary from the same employer or vendor.

3.5.2 Status of this RFC

This RFC being tested as a POC in the SSO-Bolt. See the *TODO* section below. To provide comments create an issue on the Github repository containing this documentation.

3.5.3 Limitations

A party who wants to verify the identity of the user must run a Nuts Node.

The application the user jumps to can only show the patient from the provided context.

The application the user jumps to does not get notified about logout operations from the original application. Sessions should end after closing the browser window/tab.

3.5.4 Terminology

The following terminology is based on the OAuth 2 specification.

Client Application

The application the users starts the jump from.

Resource server

The application (a protected resource) the user is trying to jump to.

JWT Bearer Token

JWT encoded Bearer token contains the user's identity, subject and custodian and is signed by the acting party. This token is used to obtain an OAuth 2 access token.

Access Token

An OAuth 2 Access Token, provided by an Authorization Server. This token is handed to the client so it can authorize itself to a resource server. The contents of the token is opaque to the client. This means that the client does not need to know anything about the content or structure of the token itself.

The resource server can exchange the Access Token at its own Authorization Server for a JSON document containing all the needed attributes.

Authorization server

The Authorization server checks the user's identity and credentials and creates the access token which should be used during the jump. The authorization server is trusted by the resource server. The resource server can exchange the access token for a JSON document with the user's identity, subject, custodian, validity and scopes. This mechanism called token introspection and is is described in rfc7662

Request Context

The context of a request identified by the access token. The access token refers to this context. The context consists of the Custodian, Actor, patient and scope of the request like, in this case, a SSO. A context can be retrieved by performing token inspection.

3.5.5 Mechanics

This document builds on Login Contracts and the RFC on Session Tokens.

We have two applications: The client application, and the resource server. Both run their own Nuts Node (could be the same node). The client application collects information about the user and patient, acquires an access token by posting a JWT Bearer Token to the authorization server (this can be the Nuts node or a custom implementation). With this access token the user jumps to the resource server. The resource server retrieves the request context by a method called token introspection.

SSO Steps

Let's look at this step by step:

1. The User loads a page in the client application in the context of a certain subject (patient).
2. The client application checks rights and settings to see if this user is allows to jump (local policy)
3. The client application checks if the subject has any external custodians (care providers). This subject <-> custodian relationship should be registered locally.
4. For each custodian the client application checks if it has a jump endpoint The endpoint is described in the *Nuts registry API*

```
$ curl --location --request GET 'actors.nuts-node.local/api/endpoints?
↪orgIds=urn:oid:2.16.840.1.113883.2.4.6.1:00000001&type=urn:oid:1.3.6.1.4.1.
↪54851.1:nuts-sso&strict=true'

 [{
   "URL": "http://sso.nootenboom.local",
   "endpointType": "urn:oid:1.3.6.1.4.1.54851.1:nuts-sso",
   "identifier": "7b8f7852-d218-4242-8406-39cf6abcde58",
   "properties": {
       "authorizationServerURL": "https://nuts.custodian.test"
   },
   "status": "active"
 }]
```

5. The client application renders a SSO button
6. The User clicks the button
7. The client application collects the users identity using a login contract. If not already present, it lets the user sign one using IRMA To create an IRMA session, make a call to the Nuts Auth server as described in the *Nuts Auth API*

```
$ curl --location --request POST 'actors.nuts-node.local/auth/contract/session' \
 --header 'Content-Type: application/json' \
 --data-raw '{
    "type": "BehandelaarLogin",
    "language": "NL",
```

(continues on next page)

(continued from previous page)

```
    "version": "v1",
    "legalEntity": "Zorggroep Nuts"
 }'

 {"qr_code_info":{"irmaqr":"signing","u":"https://yourdomain.test/auth/irmaclient/
↪session/L6onpBhXyzHnE5bVkipH"},"session_id":"OgU0be4mkKr6bzsArJQr"}
```

After the user signs the contract, retrieve the Identity Token:

```
$ curl --location --request GET 'actors.nuts-node.local/auth/contract/session/
↪OgU0be4mkKr6bzsArJQr'

{"disclosed":[{"identifier":"irma-demo.nuts.agb.agbcode","rawvalue":"00000007",
↪"status":"PRESENT","value":{"":"00000007","en":"00000007","nl":"00000007"}}],
↪"nuts_auth_token":"eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJpc3MiOiJudXRzIiwibnV0c19zaWduYXR1cmUiOnsiSXJtYUNvbnRyYWN0Ijp7IkBjb250ZXh0IjoiaHR0cHM6Ly9pcm
↪Sj7KqJDSYSD6Wjl48B-guNQoJNwpi-
↪I9oy7JD4fhyUvYfvapLr2tCyxg5auY48h9Iwz3j2E71kQ242Nj7VgINqC94aUFpBBp79v9aDC3p-
↪Sbr3RCZ2aiXGAmxhN8xerR0ETedhAeNZxFNLjkBlXhDxNHcDji11B_5mkQjTmymg_30x1CGQlJfXa_
↪l31TPoSrPWGVxu3wWYKThTK1tpRzC_
↪f9aTVHEvpFgqGLAgzY2BMNY7VwqXXMyRfrAAe2n8foiSA8lSVAa47CJWx0-
↪4svWadcPBkwp1DgwxLoMKDceNy2ZY12kWYpHtwVgUdLq6Y7nv_As66ui0f06yMhTtRlEQ",
↪"proofStatus":"VALID","status":"DONE","token":"OgU0be4mkKr6bzsArJQr","type":
↪"signing"}
```

8. Construct a JWT Bearer token by submitting the Custodian, Actor, subject and Identity Token to the jwtbearertoken endpoint:

```
$ curl --location --request POST 'actors.nuts-node.local/auth/jwtbearertoken' \
  --header 'Content-Type: application/json' \
  --data-raw '{
     "custodian": "urn:oid:2.16.840.1.113883.2.4.6.1:00000001",
     "actor": "urn:oid:2.16.840.1.113883.2.4.6.1:00000000",
     "subject": "urn:oid:2.16.840.1.113883.2.4.6.3:999999990",
     "identity": "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJpc3MiOiJudXRzIiwibnV0c19zaWduYXR1cmUiOnsiSXJtYUNvbnRyYWN0Ijp7IkBjb250ZXh0IjoiaHR0cHM6Ly9pcm
↪Sj7KqJDSYSD6Wjl48B-guNQoJNwpi-
↪I9oy7JD4fhyUvYfvapLr2tCyxg5auY48h9Iwz3j2E71kQ242Nj7VgINqC94aUFpBBp79v9aDC3p-
↪Sbr3RCZ2aiXGAmxhN8xerR0ETedhAeNZxFNLjkBlXhDxNHcDji11B_5mkQjTmymg_30x1CGQlJfXa_
↪l31TPoSrPWGVxu3wWYKThTK1tpRzC_
↪f9aTVHEvpFgqGLAgzY2BMNY7VwqXXMyRfrAAe2n8foiSA8lSVAa47CJWx0-
↪4svWadcPBkwp1DgwxLoMKDceNy2ZY12kWYpHtwVgUdLq6Y7nv_As66ui0f06yMhTtRlEQ"
  }'

  {"bearer_token":"eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJhdWQiOiJodHRwczovL3RhcmdldF90b2tlbl9lbmRwb2ludCIsImV4cCI6MTU4MTQxMzQwMiwiaWF0IjoxNTgxNDExNj
↪TF8lmmSQ4WkznMXLmh6JTA0cYwSrKoKd_
↪yK7jKzweyAZhIGv9tmxGASZ7cIg9495U9SsyGVSQUpvY0gMIYLIRENzUJ1rUCS1kYSDrIvp13DRwGrz74f7SAp8hQXer1R
↪A9bSaoWoko8IIV-Tvo1XMHLhqV-msQig5Q-IFoYtsBkdhdDBSaDEy9-
↪LJcjK5eU0Ymc781KRz5usdz3ta6QMfqfg_Ypx2NuID5bJg0Mcnw6nooMreQl7lgO_
↪clyJGfUdS0v2A7pCXi6Vc2c9zRxdNKJSLd65odDrvfhcHGNlaGZrMn07phmT13YR0e4rBAV1tkapERp23Q
↪"}
```

9. The client application requests a access token providing the JWT Bearer token This is described in the *RFC: Access Tokens*

```
$ curl --location --request POST 'https://custodians.nuts-node-local/auth/
↪accesstoken' \
  --header 'Content-Type: application/x-www-form-urlencoded' \
  --header 'X-Nuts-LegalEntity: Demo EHR' \
  --data-urlencode 'grant_type=urn:ietf:params:oauth:grant-type:jwt-bearer' \
  --data-urlencode 'assertion=eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJhdWQiOiI2M2RkYTg5Mi03ZDRkI
↪JkYSWcFgklkX8WHaNndbaFzSkG3Tu84jBaGl9qxyTBh5mgntiWlAfRxmp09U9Hhnr3Dl1ly-
↪nAPRmr2SjOIEbr736L9p9khKsLPZW5kllTeugzOwyZDHzpcvhlcQN_7_
↪XpF2E1yteVMkq91210GkekMRfAzbf9BXTGLaw'

  {
     "access_token": "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJlbWFpbCI6Im1haWxAc3ZhbmRlcnZlZ3QubmwiLCJleHAiOjE1ODI2Mjg3ODMsImZhbWlseV9uYW1lIjoiQnJ1aWpuIi
↪Ijux1Y0ASChDNuxPd7yXoaNwnaqnkJt9JYGAkOAAYyivEQW55H1GAlHsqIHy7h-VAIKB9qJLo_
↪Dumgpwu-oi_-ETjlO64Id-FDieUzSUgRAwlx3m0n1cym7072BRAoRnhFtP-ncYyxx-vI-
↪cd7UtPxVIA9adhma2768NnGAvnrSKP2Q5xao9GuBjjD3kD8XFpTX8MN-
↪u695HKH8fYTNZ2Hdtry14sTbaDnJu9CTT_lZDp_9Y9u9nwmjfjgLb_
↪pla2Q2LUFaFasFRerX6s32XulXt0WuLZt-XXFLDlc5u18UwI3koXM61-
↪VnFJ1nUsb2lq00nJyIUXi3pdR20mO1zsA",
     "expires_in": 0,
     "token_type": ""
  }
```

10. The client application redirects the user to the jump url with the access token in the URL

```
HTTP/1.1 302 Found
Location: http://sso.nootenboom.local?access_
↪token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.
↪SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

11. The resource server retrieves all information from the access token by performing token introspection

    1. Retrieve the information from the access token: rfc7662

       If the field active is equal to *true* then the access token is valid and the resource server can proceed.

```
$ curl --location --request POST 'custodians.nuts-node.local/auth/token_
↪introspection' \
  --header 'Content-Type: application/x-www-form-urlencoded' \
  --data-urlencode 'token=eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJlbWFpbCI6Im1haWxAc3Zh...
↪46b2lkOjIuMTYuODQwLjEuMTEzODgzLjIuNC42LjE6MDAwMDAwMDAifQ.rtBqB5kuWXyIA_
↪MimIyBS9OSREyf4lRb00xm8v7qPG1TYkI_PXj4p_
↪Tz2Cp4a1vwPfQag5z77eqxcjqcOUQx9Xdm44nw8seQ-
↪GqYLrf8czkSCGTNg7MuwmjuvSD2GCPfMqzYvLOo8PPBzmwthZNmBKUGuNB9Cst_
↪y6y6ShVbl3tyL9ZBabpAJwptXVg5oZBG8ANObCiruR4uf3hvm_
↪Ci3upR4xKf6U2yj55eASVEGicIdLQu-zVhKmU-1G8xLjIxD6iVqpAwH5rmcbI8nt0B-
↪9h33rbB59xW3NY4WCJ7YvplgTlOUUUP3jE2NhMmZT_Yxq00DqRweS9pr5wAdo4k6w'

  {
      "active": true,
      "aud": "",
      "exp": 1582628499,
      "iat": 1582627599,
      "iss": "urn:oid:2.16.840.1.113883.2.4.6.1:00000001",
      "name": "Willeke de Bruijn",
      "family_name": "Bruijn",
```

(continues on next page)

(continued from previous page)

```
    "prefix": "de",
    "given_name": "Willeke",
    "email": "w.debruijn@example.org",
    "scope": "nuts-sso",
    "sid": "urn:oid:2.16.840.1.113883.2.4.6.3:999999990",
    "sub": "urn:oid:2.16.840.1.113883.2.4.6.1:00000000"
}
```

12. The resource server serves the requested page to the user respecting the context from the access token.

Endpoints

The sso registry endpoint should use the following *endpointType*

```
urn:oid:1.3.6.1.4.1.54851.1:nuts-sso
```

The properties object of the endpoint should contain the *authorizationServerURL*. This is the location of the OAuth 2.0 authorization server of this endpoint. A complete registry entry can look like this:

```
{
    "URL": "http://localhost:80",
    "endpointType": "urn:oid:1.3.6.1.4.1.54851.1:nuts-sso",
    "identifier": "7b8f7852-d218-4242-8406-39cf6abcde58",
    "properties": {
        "authorizationServerURL": "http://bundy-nuts-service-space:1323"
    },
    "status": "active"
}
```

3.5.6 Setup a development network

This is all tested on MacOS and Ubuntu 19. On Windows 10 you can use the Linux bash shell. Let us now if you run into problems.

To run on Linux (Ubuntu) you'll need the following prerequisites:

- "jq" package: `$ sudo apt-get install jq`
- Rootless Docker daemon (otherwise the shell scripts will create files under *root* user rather than your own): https://docs.docker.com/engine/security/rootless/

Background

We will start up 2 nuts nodes with 3 care organizations. We will register a consent for patient Luuk from the general practitioner (Huisartsenpraktijk Nootenboom) to the long term care organization (Verpleeghuis De Nootjes). Now, people from Verpleeghuis De Nootjes can SSO to the EHR of the general practitioner.

What you'll need

The irma app installed on your phone https://irma.app/

Several IRMA (demo) credentials loaded in the IRMA App

From the demo municipality: first name, last name full name, initials, prefix https://privacybydesign.foundation/attribute-index/en/irma-demo.gemeente.personalData.html

An real email attribute https://privacybydesign.foundation/uitgifte/email/

A (free) ngrok account to allow incoming request from the IRMA app to your local machine https://ngrok.com

A fairly recent docker and docker-compse version (Make sure your docker has enough memory > 7gb)

Getting started

Follow the steps as provided in the *Setup a local Nuts network* guide

Setup the consent:

- Navigate to the EHR of Huisartsenpraktijk Nootenboom Open http://localhost:8001
- Login with irma (if you're not redirected after login, manually navigate to '/')
- Navigate to patient Luuk
- Add consent for organisation Verpleeghuis De Nootjes by opening the *Network* tab and adding a share. After adding the share, wait until the state 'pending acceptance' disappears. Now, Verpleeghuis De Nootjes may access the medical information.
- Logout

Make a SSO jump

- Open http://localhost:8000 (EHR of Verpleeghuis De Nootjes)
- Login with irma (if you're not redirected after login, manually navigate to '/')
- Navigate to Luuk
- Open the Network tab
- Click on the SSO link
- Verify you are back in the EHR of Huisartsenpraktijk Nootenboom in the context of patient Luuk

---

Note: Since the consent now has been registered, next time you can start up the network without Corda nodes by running ./start-network.sh ehr

---

3.5.7 TODO

The access token should be opaque to the client application. In a stateless token (the current implementation of the Nuts node) the token is a JWT containing the BSN of the subject. The token should be encrypted. This is currently not the case. This is fine for DEMO purposes, but should be fixed for production. Since the token is used in a GET request, it can be recorded by the browser and the server logs.

Architecture and protocol proposals.

CHAPTER 4

Getting Started With Nuts

4.1 Setup a local Nuts network

Since Nuts is a distributed network, every party in the network runs its own node. A node consist of 3 parts:

- The nuts-go application which behaves as the main access point for vendors
- A Corda application which contains all the logic for signing and distributing patient consents
- A bridge application between the Corda application and the nuts-go application

All three applications are containerized and can be found on Docker Hub.

The easiest way for starting up a local development network is by using the docker-compose configuration.

This requires Docker to be installed.

Useful links	Description
Docker	Introduction into docker
Docker compose	Introduction into docker-compose

4.1.1 Checkout the nuts network local repository

At the time of writing v0.13.0 is the latest version. Make sure to use a stable version corresponding to this version of the documentation.

```
$ git clone https://github.com/nuts-foundation/nuts-network-local.git
Cloning into 'nuts-network-local'...
remote: Enumerating objects: 129, done.
remote: Counting objects: 100% (129/129), done.
remote: Compressing objects: 100% (86/86), done.
remote: Total 129 (delta 62), reused 98 (delta 33), pack-reused 0
Receiving objects: 100% (129/129), 286.29 KiB | 1.04 MiB/s, done.
Resolving deltas: 100% (62/62), done.
```

(continues on next page)

(continued from previous page)

```
cd nuts-network-local
$ git checkout tags/0.13.0
```

4.1.2 Inspect the package

This project is mainly a `docker-compose.yml` file, a cordapp and node configuration files.

The first node is called bundy, the second dahmer (any resemblance to serial killers is purely coincidental). A little trick to keep the nodes apart: they are in alphabetical ordering and so are the port numbers etc.

When you inspect the docker-compose.yml you will see two nodes with its 3 applications and a notary.

Each node has its own configuration directory in the config/node_name folder e.g. dahmers config. It contains a private key, the bridge config (application.properties) and the nuts-go config (nuts.yaml) file. Documentation about all these configuration parameters can be found in the *Configuration section*

The registry config is shared between nodes and contains the addresses and public keys of both nodes. You are encouraged to inspect these files.

4.1.3 Generate the corda nodes

To create a network of trust, Corda uses a bootstrapper tool. This tool must be downloaded because it's to large to keep in version control. For more information about the bootstrapping process see the corda docs: https://docs.corda.net/docs/corda-os/4.4/network-bootstrapper.html We provided a script to download and run the tool.

```
$ ./bootstrap-corda.sh
Nuts network bootstrapper with Corda apps v0.13.0
WARNING: This script removes all existing corda nodes and generate new ones.
Do you want to continue? [Yes/No]Yes
removing all nodes (if any)
download new cordapps of version 0.13.0
downloading corda network boostrapper
running bootstrapper (this may take a while)
Bootstrapping local test network in /opt/app
Generating node directory for dahmer
Generating node directory for bundy
Generating node directory for notary
Nodes found in the following sub-directories: [notary, dahmer, bundy]
Found the following CorDapps: [flows-0.13.0.jar, contract-0.13.0.jar]
Copying CorDapp JARs into node directories
Waiting for all nodes to generate their node-info files...
... still waiting. If this is taking longer than usual, check the node logs.
Distributing all node-info files to all nodes
Loading existing network parameters... none found
Gathering notary identities
Generating contract implementations whitelist
New NetworkParameters {
      minimumPlatformVersion=6
      notaries=[NotaryInfo(identity=CN=nuts_corda_development_notary, O=Nuts,␣
↪L=Groenlo, C=NL, validating=false)]
      maxMessageSize=10485760
      maxTransactionSize=524288000
      whitelistedContractImplementations {
```

(continues on next page)

(continued from previous page)

```
      }
      eventHorizon=PT720H
      packageOwnership {

      }
      modifiedTime=2020-04-01T15:08:48.560Z
      epoch=1
  }
Bootstrapping complete!
done
```

4.1.4 Populate Nuts registry

The Nuts registry contains identities of vendors and care organizations and endpoints of resource servers. In order to make use of the nodes the registry must be populated. For this a script is provided:

```
$ ./setup-network-registry.sh
```

You are asked to enter the names of two software vendors. This makes it easier to implement a use-case. The script generated a lot of registry events. Take a look in config/registry/events.

4.1.5 Starting up the nodes

The Nuts node needs to accept some incoming connections from the IRMA app. To make this easier, we provided a script that boots up ngrok and puts the endpoints in the nuts config. The script will ask for a ngrok token during first boot. For this you will need a free ngrok account. Get your token at https://dashboard.ngrok.com/auth.

Note: It may take some time (up to ~4 minutes) for all nodes to be booted up. If you see errors about `Cannot connect to server(s)` just wait a little longer

---

```
$ ./start-network.sh
```

Congratulations!! You just booted a full Nuts network on your local machine :)

4.1.6 Demo EHR

To make interacting with Nuts a bit more fun we added a Demo EHR. There are 3 care organizations available on the addresses:

Care provider name	Address	Corda Node
Verpleeghuis de Nootjes	http://localhost:8000	Bundy
Huisartsenpraktijk Nootenboom	http://localhost:8001	Dahmer
Medisch Centrum Noot aan de Man	http://localhost:8002	Dahmer

To register a consent, choose a patient who is know by both organizations. Luuk Meijer is such a patient who is known to Verpleeghuis de Nootjes and known to Huisartsenpraktijk Nootenboom. Go to the patient, click in the network tab and Add a consent. This should take less than a minute. You can see the progress by going to the patients list and scroll down tot the Transactions section.

4.1.7 Record your first consent using the API

To see if everything is set up correctly, we will create a consent request by posting to the consent-logic api. More details about the api and its endpoints can be found *here*

```
$ curl -X POST \
http://localhost:11323/api/consent \
-H 'Content-Type: application/json' \
-d '{
  "subject": "urn:oid:2.16.840.1.113883.2.4.6.3:99999990",
  "custodian": "urn:oid:2.16.840.1.113883.2.4.6.1:12345678",
  "actor": "urn:oid:2.16.840.1.113883.2.4.6.1:87654321",
  "performer": "urn:oid:2.16.840.1.113883.2.4.6.1:00000007",
  "records": [{
    "consentProof": {
      "ID": "11112222-2222-3333-4444-555566667777",
      "title": "Toestemming inzage huisarts.pdf",
      "URL": "https://some.url/path/to/reference.pdf",
      "contentType": "application/pdf",
      "hash": "string"
    },
    "period": {
      "start": "2019-05-20T17:02:33+10:00",
      "end": "2019-11-20T17:02:33+10:00"
    },
    "dataClass": [
      "urn:oid:1.3.6.1.4.1.54851.1:MEDICAL"
    ]
  }]
}'
```

You can check the status the process by querying the events endpoint of the event-store:

```
$ curl -X GET  http://localhost:11323/events
{"events":[{"consentId":"7b8cf879-6d9f-4f62-b31e-165848ec5677","externalId":
↪"13bf4c28d334d712c7297613cffc97d935c9972897d6fe678fc9f3ad8e354ada",
↪"initiatorLegalEntity":"","name":"completed","payload":
↪"eyJjb25zZW50SWQiOnsiZXh0ZXJuYWxJZCI6IjEzYmY0YzI4ZDMzNGQ3MTJjNzI5NzYxM2NmZmM5N2Q5MzVjOTk3Mjg5N2Q2Zm
↪","retryCount":0,"uuid":"8d5a8857-064e-4cbd-b749-d17298335fcf"}]}
```

The `"name":"completed"` tells you the consent got successfully distributed over both nodes. If you don't believe it, checkout the event on dahmer by performing the same request on address `http://localhost:21323/events`.

To check consents for the combination of actor and subjec you can perform the following query to the consent-store:

```
$ curl -X POST \
http://localhost:11323/consent/query \
-H 'Content-Type: application/json' \
-d '{
"actor": "urn:oid:2.16.840.1.113883.2.4.6.1:87654321",
"subject": "urn:oid:2.16.840.1.113883.2.4.6.3:99999990"
}'
```

There are quite a few steps to perform by the two nuts nodes before the consent is recorded. Check out the *state machine with all the events here*.

These events get broadcast to all parties, including your own application. That's how you get notified about new patient consents. Try to inspect the payload to see what you can expect. Hint: it's base64 encoded.

That's it. You have just booted an open source distributed health infrastructure on your computer and recorded your first consent. If anything did not work out as described above, don't hesitate to *contact us*.

4.2 Let users authenticate themselves to the Nuts network

The Nuts network is based upon encryption instead of trust. This means that instead of trusting all parties in the network to provide correct data, we ensure all data is cryptographically verifiable. This includes the identity of people requesting data of patients.

Therefor in order to make a request for data by another party, a user should provide identity information that everyone in the network can verify. For that we use the IRMA framework developed by the privacy by design foundation (a spin off of the Radboud university).

In this tutorial we will show you how to include a user interface inside your application were users can authenticate themselves using IRMA. The end result will look like this:

Fig. 1: Nuts IRMA login flow

Useful links	Description
IRMA web frontend	Repository of the Nuts irma styleguide
IRMA styleguide	Styleguide on how to embed the IRMA screens in your application
Nuts auth-js	Package for easy nuts-auth irma flow integration
Tutorial Code	All code from this tutorial can be found in this github tutorial.
ngrok	Ngrok allows your phone to connect to the local nuts service

4.2.1 Creating an empty Express JS application

For demo purposes we start with an empty generated express node.js application. If you have an existing app were you would like to implement this, just use that.

Install the express-generator first

```
$ npm install -g express-generator
/usr/local/bin/express -> /usr/local/lib/node_modules/express-generator/bin/express-
↪cli.js
+ express-generator@4.16.1
added 10 packages from 13 contributors in 0.762s
```

Make a new project directory and create an empty express project using ejs for templating and sass for CSS rendering:

```
$ md nuts-auth-tutorial
$ cd nuts-auth-tutorial
$ express --view=ejs --css=sass .

 create : public/
 create : public/javascripts/
 create : public/images/
 create : public/stylesheets/
 create : public/stylesheets/style.sass
 create : routes/
 create : routes/index.js
 create : routes/users.js
```

(continues on next page)

(continued from previous page)

```
 create : views/
 create : views/error.ejs
 create : views/index.ejs
 create : app.js
 create : package.json
 create : bin/
 create : bin/www
```

Run the npm installer to install dependencies

```
$ npm install
```

Start up the empty app

```
$ npm start
```

The empty app starts at http://localhost:3000

The code so far can be found at this github branch

4.2.2 Create login page

Ok, let's begin with making a place to handle session logic, a page to show the above IRMA UI, some logic to validate and store the token in a session and some logic to for the user to logout. We can use the already generated `/users` route to show information about the current logged in user so lets create a new route for session handling:

Listing 1: /routes/session.js

```
var express = require('express');
var router = express.Router();

router.get('/login', function (req, res, next) {
  res.render('login', {})
});

module.exports = router;
```

Listing 2: /views/login.ejs

```
<h1>Login with IRMA</h1>
```

Now register the session router in *app.js*:

```
 var indexRouter = require('./routes/index');
 var usersRouter = require('./routes/users');
+var sessionRouter = require('./routes/session');

 var app = express();
 app.use('/', indexRouter);
 app.use('/users', usersRouter);
+app.use('/session', sessionRouter);
```

Change the landing page to display a login link:

```
  <p>Welcome to <%= title %></p>
+ <a href="session/login">Click here to login</a
```

Restart the app and navigate to http://localhost:3000/session/login to enjoy the result of your hard labour.

The code for progress so far can be found at github

4.2.3 Add login html and javascript

To make it convenient to integrate Nuts with your existing application, we provided some helpful tools. We created a simple styleguide and a javascript IRMA state machine which we will now embed on our freshly created login page.

Install the two npm packages:

```
$ npm add irma-web-frontend @nuts-foundation/auth
```

First start by pasting the following html which contains the instructions and Now, embed the following html on the login.ejs page. The html is taken from the Nuts irma styleguide.

Listing 3: views/login.ejs

```
<section class="nuts-login-form irma-web-form">
  <header class="header">
    <p>Login with <i class="irma-web-logo">IRMA</i></p>
    <section class="helper">
      <p>Don't know what to do here? Take a look at the <a href="https://
↪privacybydesign.foundation/irma-begin/">de
          website of IRMA</a>.</p>
    </section>
  </header>
  <section class="content">
    <section class="centered loading">
      <div class="irma-web-loading-animation"><i></i><i></i><i></i><i></i><i></i><i></i><i></i><i></
↪i><i></i><i></i><i></i></i>
      </div>
      <p>One moment please...</p>
    </section>
    <section class="centered initialized">
      <canvas id="qrcode"></canvas>
    </section>
    <section class="centered waiting-for-user">
      <div class="irma-web-waiting-for-user-animation"></div>
      <p>Follow the instructions on your phone</p>
    </section>
    <section class="centered success">
      <div class="irma-web-checkmark-animation"></div>
      <p>Success!</p>
    </section>
    <section class="centered expired">
      <p>The transaction took long</p>
      <p><a href="#" onclick="nutsLogin.start()">Try again</a></p>
    </section>
    <section class="centered cancelled">
      <p>The transaction got cancelled</p>
      <p><a href="#" onclick="nutsLogin.start()">Try again</a></p>
    </section>
    <section class="centered errored">
```

(continues on next page)

(continued from previous page)

```
35        <p>Something went wrong</p>
36        <p><a href="#" onclick="nutsLogin.start()">Try again</a></p>
37      </section>
38    </section>
39  </section>
```

On line #16 you see `<canvas id="qrcode"></canvas>`. This is the element were the qrcode will be shown.

In order to load the css and javascript from both libraries we need to configure express to serve static content

Listing 4: app.js

```
 app.use(express.static(path.join(__dirname, 'public')));
+app.use('/scripts/nuts-auth', express.static(__dirname + '/node_modules/@nuts-
↪foundation/auth/dist'));
+app.use('/style/irma-web-frontend', express.static(__dirname + '/node_modules/irma-
↪web-frontend/dist'));
```

Include them on top of the login.ejs page

Listing 5: views/login.ejs

```
+<link rel="stylesheet" href="/style/irma-web-frontend/irma-web-frontend.min.css" />
+<script src="/scripts/nuts-auth/browser.js"></script>

 <section class="nuts-login-form irma-web-form">
```

To bring our login screen to life call the `nutsAuth` service as provided. There are a few configuration options:

- **nutsAuthUrl** The external address of the nuts-node
- **qrEl** the canvas element to render the qr-code in
- **nutsAuthUrl** the address of our bundy nuts node
- **postTokenPath** the backend path to POST the acquired token to after successful login. We will create a route for this later on.

```
<script>
  nutsLogin = nutsAuth.init({
    nutsAuthUrl: "http://localhost:11323",
    qrEl: 'qrcode',
    logLevel: 'debug',
    postTokenPath: '/session/login',
    afterSuccessPath: '/users'
  })
  nutsLogin.start();
</script>
```

4.2.4 Setup your nuts-node

Since the user has to connect its phone with your Nuts server which is probably running behind a NAT or firewall, it is ofter a good idea to use a service like ngrok. The nuts node Bundy will be listening at `11323`.

```
$ ngrok http 11323
...
```

(continues on next page)

(continued from previous page)

```
Session Status                online
Region                        United States (us)
Forwarding                    https://8bc613e1.ngrok.io -> http://localhost:11323
```

The *Forwarding* address is the external location your local nuts node can be found at. In order for IRMA to work, you need to set the `auth.publicUrl` to this address: copy the https url and paste it into the config file of *bundy*.

Listing 6: nuts-network-local/config/bundy/nuts.yaml

```
verbosity: debug
address: :1323
auth:
  actingPartyCn: Demo EHR
  publicUrl: https://8bc613e1.ngrok.io
  irmaConfigPath: /opt/nuts/irma
  enableCORS: true
crypto:
  fspath: /opt/nuts/keys
registry:
  datadir: /opt/nuts/registry
events:
  zmqAddress: tcp://bundy-bridge:5563
cbridge:
  address: http://bundy-bridge:8080
cstore:
  connectionstring: /opt/nuts/sqlite/sqlite.db
```

Note: Since we are letting the browser directly connect to the nuts-node, we need to enable CORS.

Now your ready to start up the nuts-node. We use the docker-compose setup as shown by the *previous tutorial*.

Note: To save memory and startup time, for this tutorial, we only use one nuts-node. If you want to run the entire distributed network, run: *docker-compose up* instead.

```
$ docker-compose up bundy-nuts-service-space
Creating network "nuts" with the default driver
Creating nuts-network-local_bundy-nuts-service-space_1 ... done
Attaching to nuts-network-local_bundy-nuts-service-space_1
bundy-nuts-service-space_1  | time="2019-08-12T13:26:59Z" level=info msg=
↪"************************************************************"
bundy-nuts-service-space_1  | time="2019-08-12T13:26:59Z" level=info msg=
↪"************************** Config ****************************"
bundy-nuts-service-space_1  | time="2019-08-12T13:26:59Z" level=info msg="address
↪                          :1323"
bundy-nuts-service-space_1  | time="2019-08-12T13:26:59Z" level=info msg="configfile
↪                          /opt/nuts/nuts.yaml"
bundy-nuts-service-space_1  | time="2019-08-12T13:26:59Z" level=info msg="auth.
↪actingPartyCn                  Demo EHR"
bundy-nuts-service-space_1  | time="2019-08-12T13:26:59Z" level=info msg="auth.
↪address                        localhost:1323"
bundy-nuts-service-space_1  | time="2019-08-12T13:26:59Z" level=info msg="auth.
↪enableCORS                     true"
bundy-nuts-service-space_1  | time="2019-08-12T13:26:59Z" level=info msg="auth.
↪publicUrl                      https://8bc613e1.ngrok.io"
```

(continues on next page)

(continued from previous page)

```
...
bundy-nuts-service-space_1  | time="2019-08-12T13:26:59Z" level=info msg=
↪"************************************************************"
bundy-nuts-service-space_1  | time="2019-08-12T13:26:59Z" level=info msg="irma␣
↪baseurl: https://8bc613e1.ngrok.io/auth/irmaclient"
bundy-nuts-service-space_1  | time="2019-08-12T13:27:02Z" level=debug msg="enabling␣
↪CORS"
...
bundy-nuts-service-space_1  |  http server started on [::]:1323
```

4.2.5 Try a first login

In order to login with IRMA, you need the app on your phone and make sure to collect a demo agb code from the IRMA attribute index Choose a random agb code with 8 digits. You can leave the *role* field empty.

Now it's time to test if everything works together. Restart your webserver and navigate to localhost:3000/session/login. If everything is alright you will see a qr-code just like in the animation on top of this tutorial page. If not, check the console log of your browser.

After scanning the qr-code you'll see a contract text on your phone which gives the Demo EHR website permission to use the nuts network for 1 hour. After signing the contract with the demo agb code, our demo EHR will redirect to the /user endpoint. This is the subject of the next session of this tutorial.

The example code so far can be found on the github repo

4.2.6 Handling a login token

During the IRMA disclosure/signing session, the frontend is polling the nuts node for status changes. After the user successfully discloses its agb code an IRMA signature gets returned in the status update. Now the frontend will post this signature to the `postTokenPath` of our Demo EHR web application.

When our backend receives the token, it needs to check that it is valid. The npm package we already installed, contains a simple API wrapper. Now, lets create that API endpoint which accepts a POST containing shall we?

The body of the post consists of a hash with only one key: `` `nuts_auth_token ``. The value is a *base64* encoded IRMA signature. The decoded contents is not really that interesting for us, but it is for our nuts node! With this token our node can determine the user who signed the contract, and if the contract is still valid.

Listing 7: routes/session.js

```
1  const { nutsAuthClient } = require('@nuts-foundation/auth');
2
3  router.post('/login', async function (req, res, next) {
4    const nutsToken = req.body.nuts_auth_token
5
6    const client = nutsAuthClient('http://localhost:11323', 'Demo EHR');
7    const validationResponse = await client.validateToken(nutsToken);
8
9    const sessionIsValid = validationResponse.validation_result === "VALID";
10
11   if (!sessionIsValid) {
12     res.status(403).send("token invalid");
13     return;
14   }
15
```

(continues on next page)

(continued from previous page)

```
  // Store token in session information for one hour
  // Note: the cookie should be signed in production environments
  res.cookie('nutstoken', nutsToken, {maxAge: 60 * 60 * 1000});
  res.cookie('agb', validationResponse.signer_attributes['irma-demo.nuts.agb.agbcode
↪'], {maxAge: 60 * 60 * 1000});

  res.status(200).send(validationResponse);
});
```

This will take the token from the post body and post it to the the nuts-auth server.

When the UI gets back the 200 it forwards the user to the `/users` page. Lets make that one too:

Listing 8: routes/users.js

```
/* GET users listing. */
router.all('/', function(req, res, next) {
  // check if the cookie is set
  if ('nutstoken' in req.cookies) {
    next();
  } else {
    res.redirect('/session/login');
  }
});

// render the user page with the agb code
router.get('/', function (req, res, next) {
  const agb = req.cookies.agb;
  res.render('user', {agb});
});
```

Listing 9: views/user.ejs

```
<h1>Welcome user</h1>
<p>Your agb is: <%= agb %></p>

<a href="/session/logout">Click here to logout</a>
```

We are on a roll! Lets create the logout route as well:

Listing 10: routes/session.js

```
router.get('/logout', function (req, res, next) {
  res.clearCookie('nutstoken')
  res.clearCookie('agb')
  res.redirect('/')
});
```

You did it! You created a fresh node.js express application. Included the nuts npm packages. Added all the views and routes to show an IRMA UI. Contacted the server to validate the token and stored it in the users session. This token can be used to make request to the nuts network. A subject for a future tutorial. For now, play around with the code and try to embed this screen in your own application.

The full demo source code is available on this github branch.

4.3 Let patients record their consent

Bear with us!

4.4 Validate incoming api requests

Don't touch that dial!

Welcome to the Nuts Getting Started guide. Nuts can help you with mainly two common challenges in data exchange between Care Providers:

- Getting data from another party with consent from the patient
- Providing data to another party with consent from the patient

Some parties need to support only one use-case, others need both.

Whatever your situation is, you always need to setup your Nuts node first.

This tutorial will help you with:

- *Setting up a local Nuts network*
- *Letting a user identify itself*
- *Record a patients consent*
- *Validate incoming api request*

CHAPTER 5

Development

5.1 Nuts consent bridge development

The consent bridge is written in Kotlin and can be build by Gradle.

5.1.1 Dependencies

Since the bridge depends on Corda, Java 1.8 is needed. For the Oracle sdk, this means that your version needs to be > 1.8 update 151. This can give problems on several linux distro's. In that case use the latest OpenJDK 1.8.

The project is build with Gradle. A gradle wrapper is present in the project.

5.1.2 Generating code

Api stubs are generated by running

```
./gradlew codegen
```

5.1.3 Using packages from Github

Follow instructions from here https://help.github.com/en/github/managing-packages-with-github-package-registry/configuring-gradle-for-use-with-github-package-registry#authenticating-to-github-package-registry. Specifically set a *gpr.user* and *gpr.key* in your ~/.gradle/gradle.properties

5.1.4 Running tests

To run all tests, including integration tests, nats-streaming-server is required to be installed. Tests can be run by executing

```
./gradlew test
```

5.1.5 Building

Executable can be build by executing

```
./gradlew bootJar
```

5.1.6 Docker

To build locally

```
docker build . -f docker/Dockerfile-dev -t nutsfoundation/nuts-consent-bridge:latest-
↪dev
```

The `nutsfoundation/nuts-consent-bridge:latest-dev` docker image can be used to connect to one of the 2 consent nodes locally. Checkout nuts-network-local-development-docker for setting up a complete environment with `docker-compose`.

5.1.7 README

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

```
./generate_readme.sh
```

5.1.8 Documentation

To generate the documentation, you'll need python3, sphinx and a bunch of other stuff. See *Building the documentation* The documentation can be build by running

```
/docs $ make html
```

The resulting html will be available from `docs/_build/html/index.html`

5.2 Nuts Consent Logic development

This module is written in Go and should be part of nuts-go as an engine.

5.2.1 Running tests

Tests can be run by executing

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

5.2.2 Generating code

```
oapi-codegen -generate server,types -package api docs/_static/nuts-consent-logic.yaml␣
↪> api/generated.go
```

5.2.3 Building

This project is part of https://github.com/nuts-foundation/nuts-go. If you do however would like a binary, just use `go build`.

5.2.4 README

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

```
./generate_readme.sh
```

This script uses `rst_include` which is installed as part of the dependencies for generating the documentation.

5.2.5 Documentation

To generate the documentation, you'll need python3, sphinx and a bunch of other stuff. See *Building the documentation*
The documentation can be build by running

```
/docs $ make html
```

The resulting html will be available from `docs/_build/html/index.html`

5.3 Nuts consent store development

The consent store is written in Go and should be part of nuts-go as an engine.

5.3.1 Dependencies

This projects is using go modules, so version > 1.12 is recommended. 1.10 would be a minimum. Currently Sqlite is used as database backend.

5.3.2 Running tests

Tests can be run by executing

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

5.3.3 Building

This project is part of https://github.com/nuts-foundation/nuts-go. If you do however would like a binary, just use `go build`.

The client and server API is generated from the nuts-consent-store open-api spec:

```
oapi-codegen -generate server,client,types -package api docs/_static/nuts-consent-
↪store.yaml > api/generated.go
```

Generating mocks

Mocks used by other modules, generate with:

```
mockgen -destination=mock/mock_client.go -package=mock -source=pkg/consent.go
```

Binary format migrations

The database migrations are packaged with the binary by using the `go-bindata` package.

```
NOT_IN_PROJECT $ go get -u github.com/go-bindata/go-bindata/...
nuts-consent-store $ cd migrations && go-bindata -pkg migrations .
```

5.3.4 README

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

```
./generate_readme.sh
```

This script uses `rst_include` which is installed as part of the dependencies for generating the documentation.

5.3.5 Documentation

To generate the documentation, you'll need python3, sphinx and a bunch of other stuff. See *Building the documentation*
The documentation can be build by running

```
/docs $ make html
```

The resulting html will be available from `docs/_build/html/index.html`

5.4 Nuts crypto development

The crypto module is written in Go and should be part of nuts-go as an engine.

5.4.1 Dependencies

This projects is using go modules, so version > 1.12 is recommended. 1.10 would be a minimum.

5.4.2 Running tests

Tests can be run by executing

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

5.4.3 Building

This project is part of https://github.com/nuts-foundation/nuts-go. If you do however would like a binary, just use `go build`.

The server API is generated from the open-api spec:

```
oapi-codegen -generate server,client,types -package api docs/_static/nuts-service-
↪crypto.yaml > api/generated.go
```

5.4.4 Generating mocks

```
mockgen -destination=test/mock/client.go -package=mock -source=pkg/interface.go Client
```

5.4.5 README

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

```
./generate_readme.sh
```

This script uses `rst_include` which is installed as part of the dependencies for generating the documentation.

5.4.6 Documentation

To generate the documentation, you'll need python3, sphinx and a bunch of other stuff. See *Building the documentation*
The documentation can be build by running

```
/docs $ make html
```

The resulting html will be available from `docs/_build/html/index.html`

5.5 Nuts discovery service development

The discovery service is written in Kotlin and can be build by Gradle.

5.5.1 Dependencies

Since the discovery service depends on Corda, Java 1.8 is needed. For the Oracle sdk, this means that your version needs to be > 1.8 update 151. This can give problems on several linux distro's. In that case use the latest OpenJDK sdk 1.8.

The project is build with Gradle. A gradle wrapper is present in the project.

5.5.2 Generating code

To generate the Api stubs based on the Open Api Spec:

```
./gradlew generateServerApiStub
```

5.5.3 Running tests

Tests can be run by executing

```
./gradlew test
```

5.5.4 Building

An executable can be build by executing

```
./gradlew bootJar
```

5.5.5 Running

The server can be started by executing

```
./gradlew bootRun
```

This requires some files to be present in the *keys* sub-directory. Check *Nuts discovery configuration* on how to configure the keys.

5.5.6 Docker

A Dockerfile is provided. As default it will run with dev properties and keys. This can be overriden by mounting the right dirs:

```
docker run -it \
    -v {{KEYS_DIR}}:/opt/nuts/discovery/keys \
    -v {{CONF_DIR}}:/opt/nuts/discovery/conf \
    -p 8080:8080 \
    nutsfoundation/nuts-discovery:latest
```

5.5.7 README

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

```
./generate_readme.sh
```

5.5.8 Documentation

To generate the documentation, you'll need python3, sphinx and a bunch of other stuff. See *Building the documentation*
The documentation can be build by running

```
/docs $ make html
```

The resulting html will be available from `docs/_build/html/index.html`

5.6 Building the documentation

- install python3
- install pip3 (if it doesn't install automatically)
- `pip3 install sphinx`
- `pip3 install recommonmark`
- `pip3 install sphinx_rtd_theme`
- `pip3 install rst_include`
- `pip3 install sphinx-jsonschema`
- `pip3 install sphinxcontrib-httpdomain`

5.7 Nuts engine development

Separated pieces of logic are added to the nuts service executable as *engines*. Most engines have their own git repo as well. An engine is an instance following struct:

Once defined it can be registered:

```
RegisterEngine(&engine)
```

5.7.1 Engine monitoring

Each Engine must have a diagnostic function called *Diagnostics()* which returns information on how well the engine is performing. The information can be used for monitoring and/or debugging. The **status** engine exposes this information at the */status/diagnostics* endpoint in plain text format.

```
memory usage: 256m
established connections: 20
loaded engines: status, logging
...
```

5.7.2 Standalone

It is possible to run a module without adding it to the main Nuts executable by defining a Go main function:

```
// engine instance
var e = NewMyEngine()

// the rootCmd
var rootCmd = e.Cmd
```

(continues on next page)

(continued from previous page)

```
// a new global nuts config
c := cfg.NewNutsGlobalConfig()

// ignore any config prefixes for this Cmd since it is running standalone
c.IgnoredPrefixes = append(c.IgnoredPrefixes, e.ConfigKey)

// register all commandLine options added by this engine
c.RegisterFlags(e)

// load all config from parameters into global config
if err := c.Load(); err != nil {
    panic(err)
}

// inject parameters from global config into config struct of engine
if err := c.InjectIntoEngine(e); err != nil {
    panic(err)
}

// check configuration on engine
if err := e.Configure(); err != nil {
    panic(err)
}

// execute comand
rootCmd.Execute()
```

5.8 Nuts event octopus development

The event listener is written in Go and should be part of nuts-go as an engine.

5.8.1 Dependencies

This projects is using go modules, so version > 1.12 is recommended. 1.10 would be a minimum. ZeroMQ is used for listening to the events from the nuts-consent-bridge. Follow installation from the ZMQ website: http://zeromq.org/intro:get-the-software

5.8.2 Running tests

Tests can be run by executing

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

5.8.3 Generating code

```
oapi-codegen -generate server,types -package api docs/_static/nuts-event-store.yaml >␣
↪api/generated.go
```

5.8.4 Generating Mock

When making changes to the client interface run the following command to regenerate the mock:

```
mockgen -destination=mock/mock_client.go -package=mock -source=pkg/events.go
```

5.8.5 Building

This project is part of https://github.com/nuts-foundation/nuts-go. If you do however would like a binary, just use `go build`.

The server API is generated from the nuts-consent-store open-api spec:

```
oapi-codegen -generate server,types -package api docs/_static/nuts-event-store.yaml >␣
↪api/generated.go
```

Binary format migrations

The database migrations are packaged with the binary by using the `go-bindata` package.

```
NOT_IN_PROJECT $ go get -u github.com/go-bindata/go-bindata/...
nuts-consent-store $ cd migrations && go-bindata -pkg migrations .
```

5.8.6 README

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

```
./generate_readme.sh
```

This script uses `rst_include` which is installed as part of the dependencies for generating the documentation.

5.8.7 Documentation

To generate the documentation, you'll need python3, sphinx and a bunch of other stuff. See *Building the documentation* The documentation can be build by running

```
/docs $ make html
```

The resulting html will be available from `docs/_build/html/index.html`

5.9 Adding metrics

Please follow the manual at https://prometheus.io/docs/guides/go-application/

For now we'll use `promauto` to register metrics to the prometheus registry. As convention all custom metrics should start with `nuts_`. If metrics could be interpreted as it came from multiple engines, add the engine name as prefix as well, eg: `nuts_crypto_`.

5.10 Nuts Network development

The network is written in Go and should be part of nuts-go as an engine.

5.10.1 Dependencies

This projects is using go modules, so version > 1.12 is recommended. 1.10 would be a minimum.

5.10.2 Running tests

Tests can be run by executing

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

5.10.3 Building

This project is part of https://github.com/nuts-foundation/nuts-go. If you do however would like a binary, just use `go build`.

The (internal) server and client API is generated from the open-api spec:

```
oapi-codegen -generate types,server,client -package api docs/_static/nuts-network.
↪yaml > api/generated.go
```

The peer-to-peer API uses gRPC. To generate Go code from the protobuf specs you need the *protoc-gen-go* package:

```
go get -u github.com/golang/protobuf/protoc-gen-go
```

To generate the Go server and client code, run the following command:

```
protoc -I network network/network.proto --go_out=plugins=grpc,paths=source_
↪relative:network
```

To generate the mocks, run the following commands:

```
~/go/bin/mockgen -destination=pkg/mock.go -package=pkg -source=pkg/interface.go
~/go/bin/mockgen -destination=pkg/proto/mock.go -package=proto -source=pkg/proto/
↪interface.go Protocol
~/go/bin/mockgen -destination=pkg/documentlog/mock.go -package=documentlog -
↪source=pkg/documentlog/interface.go DocumentLog
~/go/bin/mockgen -destination=pkg/documentlog/store/mock.go -package=store -
↪source=pkg/documentlog/store/interface.go DocumentStore
~/go/bin/mockgen -destination=pkg/nodelist/mock.go -package=nodelist -source=pkg/
↪nodelist/interface.go NodeList
~/go/bin/mockgen -destination=pkg/p2p/mock.go -package=p2p -source=pkg/p2p/interface.
↪go P2PNetwork
```

Binary format migrations

The database migrations are packaged with the binary by using the `go-bindata` package.

```
NOT_IN_PROJECT $ go get -u github.com/go-bindata/go-bindata/...
nuts-network $ cd migrations && go-bindata -pkg migrations .
```

5.10.4 Running in Docker

Since nuts-network forms a p2p network it's useful to be able to quickly spawn a lot of nodes. The easiest way to do so it using the provided docker-compose file:

docker-compose up

It now should start a bootstrap node and a single (non-bootstrap) node.

To expand the network, add a few nodes:

docker-compose up -d –scale node=5

5.10.5 README

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

```
./generate_readme.sh
```

This script uses `rst_include` which is installed as part of the dependencies for generating the documentation.

5.10.6 Documentation

To generate the documentation, you'll need python3, sphinx and a bunch of other stuff. See *Building the documentation* The documentation can be build by running

```
/docs $ make html
```

The resulting html will be available from `docs/_build/html/index.html`

5.11 Nuts auth development

The auth module is written in Go and should be part of nuts-go as an engine.

5.11.1 Dependencies

This projects is using go modules, so version > 1.12 is recommended. 1.10 would be a minimum.

5.11.2 Running tests

Tests can be run by executing

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

5.11.3 Generating code

```
oapi-codegen -generate server,types -package api docs/_static/nuts-auth.yaml > api/
↪generated.go
```

5.11.4 Generating Mock

When making changes to the client interface run the following command to regenerate the mock:

```
mockgen -destination=mock/mock_client.go -package=mock -source=pkg/auth.go
```

5.11.5 Building

This project is part of https://github.com/nuts-foundation/nuts-go. If you do however would like a binary, just use `go build`.

5.11.6 README

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

```
./generate_readme.sh
```

This script uses `rst_include` which is installed as part of the dependencies for generating the documentation.

5.11.7 Documentation

To generate the documentation, you'll need python3, sphinx and a bunch of other stuff. See *Building the documentation*
The documentation can be build by running

```
/docs $ make html
```

The resulting html will be available from `docs/_build/html/index.html`

5.12 Nuts consent cordapp development

The consent cordapp is written in Kotlin and can be build by Gradle.

5.12.1 Dependencies

Since the consent cordapp depends on Corda, Java 1.8 is needed. For the Oracle sdk, this means that your version needs to be > 1.8 update 151. This can give problems on several linux distro's. In that case use the latest OpenJDK 1.8.

The project is build with Gradle. A gradle wrapper is present in the project.

5.12.2 Running tests

Tests can be run by executing

```
./gradlew test
```

5.12.3 Building

Jars can be build by executing

```
./gradlew jar
```

5.12.4 Docker

To build locally

```
docker build . -f docker/Dockerfile-dev
```

The `nutsfoundation/nuts-consent-cordapp:latest-dev` docker image can be used to run 3 nodes locally. Checkout nuts-network-local-development-docker for setting up a complete environment with `docker-compose`.

5.12.5 README

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

```
./generate_readme.sh
```

5.12.6 Documentation

To generate the documentation, you'll need python3, sphinx and a bunch of other stuff. See *Building the documentation* The documentation can be build by running

```
/docs $ make html
```

The resulting html will be available from `docs/_build/html/index.html`

5.12.7 Release

Both the flows and contract libs are published to maven central (through OSS Sonatype). Before you can release and sign the jars, you need the following things:

- a valid gpg setup
- a published gpg key
- a sonatype account linked to nl.nuts

You can release libraries through:

```
./gradlew uploadArchives
```

Then go to https://oss.sonatype.org and *close* and *release* the libs. More info can be found on https://central.sonatype.org/pages/releasing-the-deployment.html.

Note: It seems signing require Oracles JVM! So openjdk won't work.

5.13 Nuts local network

5.13.1 Running with docker-compose

This repo contains a set of properties, keys, config and other files for setting up a local development environment. This is all connected together with a single `docker-compose.yml` file. You'll need to have docker and java installed.

For more extensive instructions, see the getting started guide in the docs: https://nuts-documentation.readthedocs.io/en/add-sso-rfc/pages/getting_started/local_network.html#setup-a-local-nuts-network .

In order to set up the network, a few script are included. To bootstrap the Corda nodes and gernerate the network event run:

```
$ ./bootstrap-corda.sh
$ ./setup-network-registry.sh
```

To start the full network

```
$ ./start-network.sh
```

Corda logs can be viewed inside `nodes/NODE/logs` since `nodes/NODE` is mounted in the container.

All of the Nuts docker images are build directly from code on Docker Hub. To get the latest development images, use:

```
docker-compose pull
```

Note: Local development runs without a discovery service.

Note: Running everything at a single machine can be a bit demanding since you're virtually running 3 nodes instead of 1. If things go too slow, give docker some more resources.

5.13.2 Data

All changes made to consent is persisted and available between restarts. All required data is stored under `./nodes/*`.

Example consent request that can be send to the bundy node (`localhost:11323/api/consent`)

```
{
    "subject": "urn:oid:2.16.840.1.113883.2.4.6.3:999999990",
    "custodian": "urn:oid:2.16.840.1.113883.2.4.6.1:00000000",
    "actor": "urn:oid:2.16.840.1.113883.2.4.6.1:00000001",
    "performer": "urn:oid:2.16.840.1.113883.2.4.6.1:00000007",
    "records": [{
        "consentProof": {
```

(continues on next page)

(continued from previous page)

```
            "contentType": "text/plain",
            "data": "cGRmIGRvY3VtZW50IHdpdGggc2lnbmF0dXJl"
        },
        "period": {
              "start": "2019-07-03T12:00:00+02:02",
              "end": "2020-07-01T12:00:00+02:00"
        }
    }]
}
```

5.14 Nuts service executable development

5.14.1 Dependencies

Go version => 1.13 is required.

5.14.2 Running tests

Tests can be run by executing

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

5.14.3 Building

just use `go build`.

5.14.4 README

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

```
./generate_readme.sh
```

This script uses `rst_include` which is installed as part of the dependencies for generating the documentation.

5.14.5 Documentation

The configuration options documentation is generated from the actual flags provided by the engines. When engines are updated, this documentation should be regenerated to reflect any changes in provided flags. To regenerate the configuration documentation run the following command from the project root:

```
make update-docs
```

To build the documentation, you'll need python3, sphinx and a bunch of other stuff. See *Building the documentation* The documentation can be build by running

```
/docs $ make html
```

The resulting html will be available from `docs/_build/html/index.html`

5.15 Nuts registry development

The registry is written in Go and should be part of nuts-go as an engine.

5.15.1 Dependencies

This projects is using go modules, so version > 1.12 is recommended. 1.10 would be a minimum.

5.15.2 Running tests

Tests can be run by executing

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

5.15.3 Building

This project is part of https://github.com/nuts-foundation/nuts-go. If you do however would like a binary, just use `go build`.

The server and client API is generated from the open-api spec:

```
oapi-codegen -generate types,server,client -package api docs/_static/nuts-registry.
↪yaml > api/generated.go
```

5.15.4 Generating Mocks

These mocks are used by other modules

```
mockgen -destination=mock/mock_client.go -package=mock -source=pkg/registry.go
mockgen -destination=mock/mock_db.go -package=mock -source=pkg/db/db.go
mockgen -destination=pkg/network/mock.go -package=network -source=pkg/network/
↪ambassador.go
```

5.15.5 README

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

```
./generate_readme.sh
```

This script uses `rst_include` which is installed as part of the dependencies for generating the documentation.

5.15.6 Documentation

To generate the documentation, you'll need python3, sphinx and a bunch of other stuff. See *Building the documentation*
The documentation can be build by running

```
/docs $ make html
```

The resulting html will be available from `docs/_build/html/index.html`

A good place to start when integrating with your own development environment is *Running with docker-compose*

Most of the API implemtations are generated from open-api spec. For go we use oapi-codegen. Get it via:

```
go get github.com/deepmap/oapi-codegen/cmd/oapi-codegen
```

CHAPTER 6

API specifications

6.1 Nuts Auth API

6.2 Nuts consent bridge API

6.3 Nuts consent logic API

6.4 Nuts consent store API

6.5 Nuts Crypto API

6.6 Nuts discovery Api

The *Nuts Discovery Service* consists of 3 api's: the network map and certificate api's. The certificate api's are only used in the initial setup phase of a node. To publish the node details with an electronic signature and retrieve the signed certificate from the *Nuts Discovery Service*. The network map api is used to retrieve details about all the nodes that are connected to the network. These api's are called by the Corda node and should not be called by any other logic.

6.6.1 Nuts certificate API

The *Nuts certificate API* is similar to the *Corda Certificate API* but is meant for the Nuts part of the CA tree. It's main goal is to support the Network authority in signing certificate signing requests and allowing the vendors to download them.

6.6.2 Corda Certificate API

The *Corda Certificate API* is part of the Corda node but it's seems to be lacking documentation. After reverse engineering, 2 requests could be identified. The initation process for a Corda node can be started with:

```
java -jar corda.jar initial-registration -p <password>
```

where the given password is the password of the Corda root truststore (for Java, *changeit* is used a lot). More info on how to start and configure a node can be found in node setup.

When started, the Corda node will create a couple of keystore as seen here. The node will send a certificate request to the configured *Doorman* url followed by /certificate. The return value will be an identifier which the node will use to poll the service. When the service has signed the request, the certificate is then downloaded and put in the keystore. Below are the details of the two services.

POST /certificate
Receiving endpoint for a certificate request (CSR). The body must be in PKCS10 byte format. The response is a plain text response with an identifier which can be used in the GET call.

Request Headers

- *Platform-Version* – the Corda platform version (4)
- *Client-Version* – todo
- *Private-Network-Map* – todo

Status Codes

- 200 OK – no error, body is a plain text string identifier
- 400 Bad Request – wrong format certificate signing request

GET / (*request_id*)
Retrieve the signed certificate which was requesting using the POST method. *request_id* is the returned identifier from the POST request. The body contains a zip file with 3 files: **cordaclientca.cer**, **cordaintermediateca.cer** and **cordarootca.cer** (in this order). Each file is the ASN.1 DER encoding of a X.509 certificate.

Response Headers

- *Content-Disposition* – (attachment; filename="certificates.zip")

Status Codes

- 200 OK – Certificate request has been signed, the body consists of a zip file with a list of certificates.
- 400 Bad Request – something went wrong.
- 404 Not Found – Unknown identifier or request hasn't been signed yet.

6.6.3 Network map API

As described on https://docs.corda.net/network-map.html

Fetching specific (old) network parameters and acking a new set of parameters is currently beyond the scope of the current supported modes.

POST /network-map/publish
This endpoint is called by a node during its startup phase. The node will send a serialized **SignedNodeInfo**

object which has been signed with the node private key. The *Nuts Discovery Service* will store the **SecureHash**, the unsigned **NodeInfo** and the list of **signatures**. The SecureHash will function as an index within the NetworkMap. The signatures are checked by other nodes when they download the NodeInfo for this node.

todo: move Parsing logic:

```
fun acceptNodeInfo(@RequestBody input: ByteArray) : ResponseEntity<ByteArray> {
  try {
    val signedNodeInfo = ByteArrayInputStream(input).readObject<SignedNodeInfo>()
    val hash = signedNodeInfo.raw.hash
    val nodeInfo = signedNodeInfo.verified()
    val signatures = signedNodeInfo.signatures

    nutsRegistrarService.publishNode(hash, nodeInfo, signatures)

  } catch (e: Exception) {
    logger.error(e.message, e)
  }

  return ResponseEntity.ok("".toByteArray())
}
```

resheader Content-Type application/octet-stream

statuscode 200 Ok with empty body

statuscode 500 Something went wrong

POST /network-map/ack-parameters
currently not implemented

GET /network-map
Returns the currently global active NetworkMap. All nodes that have been published and accepted by the *Nuts Discovery Service* will be in the output list. The output only consists of the node hashes and the hash of the current active network parameters. The call returns a **SignedNetworkMap** object signed with the NetworkMap private key. The cache control header is used by the node for a refresh interval.

Response Headers

- Content-Type – application/octet-stream
- Cache-Control – max-age=[X seconds]

Status Codes

- 200 OK – Ok with serialized NetworkMap
- 500 Internal Server Error – Something went wrong

GET /network-map/ (*var*)
currently not implemented

GET /network-map/node-info/ (*hash*)
Fetch the specific **NodeInfo** indicated by *hash*. The NodeInfo will be the same as published by the node. The *Nuts Discovery Service* can't manipulate this since the signatures correspond to the private key of the node. The result will be a **SignedNodeInfo** object. The original NodeInfo and signatures from the publish api are used.

Response Headers

- Content-Type – application/octet-stream

Status Codes

- 200 OK – Ok with SignedNodeInfo object

- 404 Not Found – Unknown hash

GET /network-map/network-parameters/ (*hash*)
Fetch the specific **NetworkParameters** indicated by *hash*. Currently this only returns the currently active NetworkParameters. The NetworkParameters contain:

- minimum platform version
- a list of notaries
- maximum message size in bytes
- maximum transaction size in bytes
- modified timestamp
- epoch (unknown what this does)
- a whitelist of approved contract implementation

Response Headers

- Content-Type – application/octet-stream

Status Codes

- 200 OK – Ok with SignedNetworkParameters object
- 404 Not Found – Unknown hash

6.7 Nuts registry API

REST and other

CHAPTER 7

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.

7.1 Documentation contributions

Documentation is written in Restructured Text. A CheatSheet can be found here.

You can test your documentation by installing the required components.

You first have to install python, check https://www.python.org/ on how to install python for your OS.

Note: MacOS (currently) comes with Python 2 preinstalled which has been deprecated as of January 1st, 2020. It's recommended to upgrade to Python 3 before proceeding.

Next you have to install pip. Then install the following components using **pip**:

```
pip install sphinx --user
pip install recommonmark
pip install sphinx_rtd_theme
pip install sphinxcontrib.httpdomain
pip install sphinx-jsonschema
pip install rst_include
```

For MacOS make sure the sphinx executables are added to your PATH (e.g. for Python 3.8):

    export PATH=$HOME/Library/Python/3.8/bin:$PATH

Then you can generate the documentation locally with:

```
make html
```

For small changes you might want to add the *clean* directive:

```
make clean html
```

The documentation will then be available from `_build/html/index.html`

7.2 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.

CHAPTER 8

Release notes

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

8.1 v0.15.0

This release adds the p2p registry. Github is still used as backup registry but registry updates should now be instant!

Project board: https://github.com/orgs/nuts-foundation/projects/9 Required migrations: *Migration*

8.1.1 Features / improvements

- Add Nuts network, a p2p network using grpc. (https://github.com/nuts-foundation/nuts-registry/pull/119)
- Process events received through Nuts network. (https://github.com/nuts-foundation/nuts-registry/issues/141)
- GenerateKeyPair should only overwrite existing key pair if specified. (https://github.com/nuts-foundation/nuts-crypto/issues/83)
- Added support for Prometheus format diagnostics.
- Added client interface for remote crypto administration. (https://github.com/nuts-foundation/nuts-crypto/issues/18)
- Add baseline truststore. (https://github.com/nuts-foundation/nuts-crypto/issues/61)
- Add certificate expiration monitoring. (https://github.com/nuts-foundation/nuts-crypto/issues/65)

8.1.2 Bugfixes

- Vendor ID was parsed incorrectly from <0.15 certificates. (https://github.com/nuts-foundation/nuts-registry/issues/142)
- Irma scheme didn't update for validator. (https://github.com/nuts-foundation/nuts-auth/pull/92)

8.2 v0.14.0

This release added additional information to the Corda states. This allows vendor software to diagnose any problems and have a better understanding what the current state of a consent request is.

For specific issues also see: https://github.com/orgs/nuts-foundation/projects/12

8.2.1 Upgrading from v0.13.0

Starting version 0.14 vendors and organizations will have an X.509 certificate (encoded in the JWK) associated with their key pairs. These certificates are used to identify the holder of the key pair used to sign events, which is also introduced in this version.

Action required: This migration can be performed by the Nuts node; use the newly introduced *registry verify* command with the *–fix* flag to generate key pairs (if necessary), issue certificates and sign events. **Don't forget to publish these changes to the central registry.** See *5. Verifying and fixing registry data* for the command.

See *Migration* for a complete overview of all migrations.

8.2.2 Features / improvements

- Support update events in registry. (https://github.com/nuts-foundation/nuts-registry/issues/41)
- Remove vendor Id from vendor CLI commands. (https://github.com/nuts-foundation/nuts-registry/issues/47)
- Auto generate vendor and organisation certificates. (https://github.com/nuts-foundation/nuts-registry/pull/94)
- Make HTTP client time-out configurable. (https://github.com/nuts-foundation/nuts-registry/issues/101)
- Add cancel flow for ConsentRequestState. (https://github.com/nuts-foundation/nuts-consent-cordapp/issues/16)
- Support cancellation event (https://github.com/nuts-foundation/nuts-consent-bridge/issues/7)
- Fixed readEvent() to not break Ref() calclations. (https://github.com/nuts-foundation/nuts-registry/issues/100)
- Register initiating node and legalEntity in branch. (https://github.com/nuts-foundation/nuts-consent-cordapp/issues/47)
- Allow administrators to reissue vendor and organization certificates (https://github.com/nuts-foundation/nuts-registry/issues/97)
- Remove legacy_auth_token (https://github.com/nuts-foundation/nuts-auth/issues/53)
- publish errors to normal channel to initiate cancellation. (https://github.com/nuts-foundation/nuts-consent-logic/issues/63)
- Add extra origin information for a consentRequest (https://github.com/nuts-foundation/nuts-consent-logic/pull/62)
- Removed decodeURI, not needed for latest echo and updated all modules (https://github.com/nuts-foundation/nuts-go/pull/43)
- Update event diagram with 'closed' event (https://github.com/nuts-foundation/nuts-event-octopus/pull/35)

8.2.3 Bugfixes

All bugs were fixed in 0.13.x versions.

- Missing endpoint does not result in errored event or retry. (https://github.com/nuts-foundation/nuts-consent-bridge/issues/61)
- Already uploaded attachment gave error. (https://github.com/nuts-foundation/nuts-consent-bridge/issues/52)
- Incorrect starting point of Nats subscription. (https://github.com/nuts-foundation/nuts-consent-bridge/issues/56)
- Corda connection monitoring not working properly (https://github.com/nuts-foundation/nuts-consent-bridge/issues/54)
- Not enough memory allocated for Corda in Docker image. (https://github.com/nuts-foundation/nuts-consent-cordapp/issues/42)
- Endpoint deduplication in API removing too much endpoints. (https://github.com/nuts-foundation/nuts-registry/issues/90)
- Contract validation incomplete. (https://github.com/nuts-foundation/nuts-auth/issues/56)
- Retry event doesn't increment retryCount. (https://github.com/nuts-foundation/nuts-consent-bridge/issues/53)
- DemoEHR endpoints have the wrong port (https://github.com/nuts-foundation/nuts-network-local/issues/11)
- Fix sphinx python script to allow for semantic versioning (https://github.com/nuts-foundation/nuts-documentation/issues43)
- Diagnostics shows Nats DOWN state (https://github.com/nuts-foundation/nuts-consent-bridge/issues/63)
- Diagnostics shows Corda DOWN state (https://github.com/nuts-foundation/nuts-consent-bridge/issues/64)
- Records can't be recorded twice: dup record/unique constraint (https://github.com/nuts-foundation/nuts-consent-store/issues/55)
- Registering vendor doesn't survive restarts: (https://github.com/nuts-foundation/nuts-registry/issues/103)

8.3 v0.13.0

Focus of this relase was on robustness; automatic resumption of listeners/handlers, no more lost events or duplicate events. Developer Happiness by providing scripts to setup the nuts-local-network. Adding a Demo-EHR so nuts becomes clickable. Improving the registry by making it event based and adding signatures. Provide a convenient API authorization method by providing an easy to use OAuth 2 flow which accepts IRMA identity tokens.

For specific issues also see: https://github.com/orgs/nuts-foundation/projects/8

8.3.1 Upgrading from v0.12.0

The registry now also contains vendors and since a node must be linked to a vendor, the *nuts.yaml* must now contain an *identity*. Checkout *Nuts service config* for the details.

See *Migration* for a complete overview of all migrations.

8.3.2 Features / improvements

- Added chaos testing setup which runs tests while pausing different parts of the Nuts node. (https://github.com/nuts-foundation/nuts-chaos-testing)
- Updated Corda to 4.4 (https://github.com/nuts-foundation/nuts-consent-cordapp/issues/39)
- Querying endpoints now also returns the organisation ID for the endpoint. (https://github.com/nuts-foundation/nuts-registry/issues/64)
- Added OAuth2 flow using Irma contracts as zero-knowledge-proof tokens to get access token. (https://github.com/nuts-foundation/nuts-auth/issues/31)
- Better reliability of the bridge due to better connection handling. (https://github.com/nuts-foundation/nuts-consent-bridge/issues/46)
- Added docker container health check for nuts-cordapp image. (https://github.com/nuts-foundation/nuts-consent-cordapp/issues/38)
- Added docker container health check for nuts-bridge image. (https://github.com/nuts-foundation/nuts-consent-bridge/issues/47)
- Added docker container health check for nuts-service-space image. (https://github.com/nuts-foundation/nuts-go/issues/15)
- The starting point for Corda events is now persisted so the bridge will resume after restart. (https://github.com/nuts-foundation/nuts-consent-bridge/issues/21)
- Registry entries for vendor now have a signed certificate. (https://github.com/nuts-foundation/nuts-registry/issues/24)
- Registry entries for organisation now have a signed certificate. (https://github.com/nuts-foundation/nuts-registry/issues/26)
- Registry entries are now signed. (https://github.com/nuts-foundation/nuts-registry/issues/60)
- Registry entries are validated and errors are shown when the certificate hierarchy is incorrect. (https://github.com/nuts-foundation/nuts-registry/issues/25)
- Added CLI mode to the nuts executable.
- Added *registerVendor*, *registerOrganization* and *registerEndpoint* commands tro CLI mode. (https://github.com/nuts-foundation/nuts-registry/issues/30)
- Authorization server type endpoint has been added to the registry. (https://github.com/nuts-foundation/nuts-registry/issues/44)
- Consent bridge now also publishes to the retry and error queue when things go wrong. (https://github.com/nuts-foundation/nuts-consent-bridge/issues/41)
- Added identity parameter to nuts-go config. (https://github.com/nuts-foundation/nuts-registry/issues/43)
- Added persistence to discovery service, CSR's, signed certificates, the network config and node information now survive restarts.
- Prepared for change in endpoint type identifier for bridge. (https://github.com/nuts-foundation/nuts-consent-bridge/issues/39)

8.3.3 Bugfixes

- It wasn't possible to add a legal base for data exchange when the custodian and actor were serviced by the same node. (https://github.com/nuts-foundation/nuts-consent-logic/issues/30)

- Disabled Corda auto-reconnect. According to Corda it is experimental and indeed it did not function well. (https://github.com/nuts-foundation/nuts-consent-bridge/issues/38)

8.4 v0.12.0

See github project for more details

8.4.1 Features / improvements

- Added status endpoint for consent-bridge available under /status
- Added status endpoint for service executable available under /status
- Added diagnostics endpoint for consent-bridge available under /status/diagnostics giving information about the service health. Things like connection status, disk status etc.
- Added diagnostics endpoint for service executable available under /status/diagnostics giving information about the service health. Things like connection status, disk status etc.
- Added docs about service monitoring
- JWK's are now internally used for key representation
- Added Ping flow to Corda to check if nodes can contact each other. Available via diagnostics
- Corda contract now also checks if old consent records are re-offered
- When creating a session, the existence of the given legal entity is checked
- The registry files have changed from state-based to event-based.

8.4.2 Bugfixes

- The public key JWT check was broken (nuts-foundation/nuts-auth#29)
- The return value for the consent check was wrong (nuts-foundation/nuts-consent-store#30)
- Path variables in http service are now decoded correctly (nuts-foundation/nuts-go-core#7)
- Fix for consent query when no validTo was given (nuts-foundation/nuts-consent-store#31)

8.5 v0.11.2

See github project for more details

8.5.1 Bugfixes

- Consent conversion from and to the internal FHIR record was broken due to missing namespacing. (https://github.com/nuts-foundation/nuts-fhir-validation/issues/8) Additionally the dataClass format is also checked in the consent POST call. (https://github.com/nuts-foundation/nuts-consent-logic/issues/23)
- The validity period now uses DateTime values instead of LocalDates. This is needed to end a particular consent immediately. (https://github.com/nuts-foundation/nuts-consent-cordapp/issues/32)

- Searching and checking active consent could result in the wrong answer when a newer version ended consent. (https://github.com/nuts-foundation/nuts-consent-store/issues/24)
- ValidTo is now optional in a validity period. There was a mismatch between different parts of the system.
- Searching for consent with a validAt parameter used string comparison and not date comparison. (https://github.com/nuts-foundation/nuts-consent-store/issues/22)
- RFC3339 time notation is now used for all dateTime values. https://github.com/nuts-foundation/nuts-consent-store/issues/25)

8.5.2 Upgrading from v0.11.0

Because of the corrupted dataClasses, all data has to be wiped. Both the *persistence.mv* for Corda and the sqlite DB for the consent store have to be deleted.

8.6 v0.11.0

See github project for more details

8.6.1 Features / improvements

- A version number has been added to the FHIR consent record (*FHIR consent requirements*) which is also visible in the consent-store. Currently, the API's will only return the latest version. The version is mainly for forwards compatibility and for viewing changes in consent in future releases.
- The consent-store query API has been changed to return a *PatientConsent* model instead of a *SimplifiedConsent* model , ref: *Nuts consent store API*.
- Changed consent on the level of individual FHIR resources (Patient, Observation, etc) to data classes (Medical, Social, Mental) across all modules. Mapping individual FHIR resources to and from classes is future work.
- Public keys in registry can now be stored in JWK format. All api's that request or return public keys can handle JWK format.
- Period dates in the consent store have been changed to datetime objects instead of dates. This is mainly done for when consent is withdrawn, it should not be active for the rest of the day.
- Corda has been updated to 4.3.

8.6.2 Bugfixes

- Fix incorrect return values for hash and ID in the consent-store api
- Fix usage of validAt query param on consent-store query api
- Fix period adherence in login contract creation
- Fix technical error when validating login contract

8.7 v0.10.0

See github project for more details

8.7.1 Features / improvements

- Signed JWTs with private key of requestor. This allows the custodian to check if JWT has been created by the requestor instead of being reused from another party.
- Add strictmode flag which forbids unsafe config options.
- Add IRMA schememanager config flag which allows setting demo or production attributes
- Recover events on startup
- Purge completed events at startup
- Add retry queues for failed events by a temporary cause
- Make nats subscription durable
- Updates all the modules to go 1.13, allowing for the new encapsulating errors
- Compare public keys by object instead of by string

8.7.2 Bugfixes

- Fix 500 on createConsent API call when body is incomplete / empty
- Fix nullpointer error on incorrect legalName in cordapp

CHAPTER 9

Network setup

This section describes how to setup a Nuts network and the steps that are needed for a Vendor to connect to the network.

9.1 Joining a network

At the moment of writing you can only participate in the network as a software vendor.

9.1.1 Joining as vendor

To join a network your node needs 2 certificates: one for the *Corda network* and one for the *Nuts network*. These certificates are issued by the *Nuts Network Authority* (although currently the second one is still self-signed). The bootstrap process looks as follows:

1. Generate Corda network Certificate Signing Request (CSR).
2. Submit Corda network CSR to Nuts Foundation and loading it into the Corda node.
3. Register vendor which generates a self-signed Nuts network certificate.
4. Restart the Nuts node for the changes to take effect (required for the certificate, generated by step 3 to be activated).

The diagram below illustrates the activities involved for step 3 and 4:

1. Generate Corda network CSR

To join the Corda network, your node needs a CA certificate so it can issue certificates to identify itself to other Corda nodes. This CA certificate is issued by the Nuts Network Authority through a CSR. A Corda node can generate this CSR for you. To do this, you'll need the following directory structure:

- **NODE_BASE_DIR**

- node.conf
- certificates/truststore.jks

A basic node.conf looks like this:

```
myLegalName="O=Nuts,C=NL,L=IJbergen,CN=node"
emailAddress="info@nuts.nl"
devMode=false
networkServices {
    doormanURL = "http://discovery:8080/doorman"
    networkMapURL = "http://discovery:8080"
}
p2pAddress="your_p2p_address:7886"
trustStorePassword="changeit"
keyStorePassword="cordacadevpass"
```

It's important to change `myLegalName`. Enter the correct password for the trustStore under `trustStorePassword`, this should have been published next to the `truststore.jks` file. `keyStorePassword` should not be changed. The `networkServices` will not resolve but this is not an issue for now. All extra settings are not important for now.

The `truststore.jks` file has to be downloaded and placed in the certificates directory.

When you're happy with your config run:

```
docker run -it \
    -v {{NODE_BASE_DIR}}/node.conf:/opt/nuts/node.conf \
    -v {{NODE_BASE_DIR}}/certificates:/opt/nuts/certificates \
    nutsfoundation/generate-csr:latest
```

where `NODE_BASE_DIR` points to the directory where the former files live.

Note: After the creation of the CSR and before you run the node. The `networkService` part of the config has to be removed (while running without the discovery service). Also `devMode` should be set to true.

---

2. Submitting Corda network CSR

When the docker container finishes running a `csr.pem` file will have been placed in the `NODE_BASE_DIR` directory. Send this file to the Nuts foundation.

When you receive a zip package, you can continue with *Loading keys*.

3. Registering vendor

For a detailed guide on how to register your vendor, Please refer to *1. Registering a vendor*.

9.2 Approving a vendor

Network administrator action

As a network administrator, it's your job to approve requests from a vendor. Requests will be sent in the form of PEM-encoded *certificate signing requests*.

9.2.1 Check source

The first thing to do is to check the medium that has been used to send you the CSR. Can you trust it? Does it have end-2-end encryption? Next: can you trust the person that has sent you the CSR?

9.2.2 Check contents

If all is correct, the applicant used the provided Docker images to generate the request. To be sure, you can use **openssl** to inspect the contents.

9.2.3 Approve the request

First make sure the discovery service is running with the correct keys! Check *Running the network discovery service*.

The request can be approved by running the following:

```
scripts/approve.sh PATH_TO_PEM [DISCOVERY_BASE_URL]
```

The `PATH_TO_PEM` is the path to the vendor CSR. The `DISCOVERY_BASE_URL` is optional and should point to the base URL of the running discovery service. Default: *http://localhost:8080*.

The result of the script is a zip file with:

- the certificate chain
- the latest network-parameters

This file can be given to the vendor which should be able to start their node.

> **Warning:** Before vendors can join, a Notary must have been approved and added, otherwise the networParameters change and have to be redistributed.

9.3 Running the network discovery service

Network administrator action

In order to be able to support the network by signing new requests and generating the *networkParameters*, the **discovery** service is needed. https://github.com/nuts-foundation/nuts-discovery

The easiest way to run the service is to use a docker image and mount the keys and configuration files. The docker image is hosted on https://hub.docker.com/repository/docker/nutsfoundation/nuts-discovery

9.3.1 Configuration

Below is an example application.properties file that can be used with docker. The `db`, `keys` and `conf` dirs need to be mounted. (assuming application.properties is placed in the conf dir)

```
server.port=8080
spring.jackson.serialization.FAIL_ON_EMPTY_BEANS=false

# JPA
spring.datasource.url=jdbc:h2:file:/opt/nuts/discovery/db/discovery_db
```

(continues on next page)

(continued from previous page)

```
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

# migrations
spring.flyway.locations=classpath:db/migration/{vendor},classpath:db/migration/common

# app specific
nuts.discovery.CordaRootCertPath = /opt/nuts/discovery/keys/root.crt
nuts.discovery.intermediateKeyPath = /opt/nuts/discovery/keys/doorman.key
nuts.discovery.intermediateCertPath = /opt/nuts/discovery/keys/doorman.crt
nuts.discovery.networkMapCertPath = /opt/nuts/discovery/keys/network_map.crt
nuts.discovery.networkMapKeyPath = /opt/nuts/discovery/keys/network_map.key

nuts.discovery.autoAck = false
```

The docker command would then be:

```
docker run -it \
    -v {{KEYS_DIR}}:/opt/nuts/discovery/keys \
    -v {{DB_DIR}}:/opt/nuts/discovery/db \
    -v {{CONF_DIR}}:/opt/nuts/discovery/conf \
    -p 8080:8080 \
    nutsfoundation/nuts-discovery:latest java -jar /opt/nuts/discovery/bin/nuts-
↪discovery.jar --spring.config.location=file:/opt/nuts/discovery/conf/application.
↪properties
```

Warning: wherever `/opt/nuts/discovery/keys` and `/opt/nuts/discovery/db` point to. Make a backup and don't lose the contents. Getting everything restored is a royal pain in the ass.

9.4 Loading keys

Vendor action

You can unzip the file given by the network operator. This should result in the following structure:

- **DIST_DIR**
  - certs/
  - conf/sslkeystore.conf
  - network-parameters

To load the certificates and generate a key for the TLS connection, you'll need to startup a docker container. This requires the same directory structure as earlier (when generating the CSR):

- **NODE_BASE_DIR**
  - node.conf
  - certificates/truststore.jks

It's important to point to the exact same directory since the generated private key exist in `NODE_BASE_DIR/certificates/nodekeystore.jks`

Then run the following command:

```
docker run -it \
    -v {{NODE_BASE_DIR}}:/opt/nuts/node \
    -v {{DIST_DIR}}:/opt/nuts/dist \
    nutsfoundation/load-certificate:latest
```

Where `NODE_BASE_DIR` points to the directory where the node.conf file exists (and the certificates directory). The `DIST_DIR` points to the root of the files extracted from the zip (see above).

It is expected that all passwords are `cordacadevpass`

9.5 Creating a new network

Network administrator action

Creating a new network begins with generating the root keys. These are needed to enable the Corda part of the network.

9.5.1 Requirements

The only thing needed is docker. A docker image is provided that will generate all the needed keys. Once the keys are generated, the root key will have to be kept safe and offline. You can print it or put in on a portable storage device and put this in a safe at a bank, poor it in concrete or shoot it to outer space depending on your requirements.

9.5.2 Docker image creation

Before generating the keys, it's wise to validate the docker image first. The best way to do this is to generate the image yourself using the dockerfile. The Github repo at https://github.com/nuts-foundation/nuts-network-setup contains all needed info. If you do not tag your image with *nutsfoundation/generate-root-key*, replace that name with your name when needed.

Note: If you want to do everything offline, you'll have to pull the repo and create the image first and then go offline.

9.5.3 Generating keys

Once an image is on the desired machine and you've gone offline. Run the following docker command:

```
docker run -it \
    -v LOCAL_PATH:/opt/nuts/keys
    nutsfoundation/generate-root-key:latest NETWORK_NAME COUNTRY LOCALITY
```

Where `LOCAL_PATH` is the path where you want your keys to be stored, `NETWORK_NAME` is the desired name of your network, `COUNTRY` is the two-letter country code and `LOCALITY` is the city of registration. During the process, you'll be asked for a password. This password will be the password for the `truststore.jks` and will have to be provided to the network participants next to the `truststore.jks` file. Reply with `yes` when asked if you want to trust the specific certificate.

9.5.4 Storing the output

The container will generate a bunch of files. The `root.key` has to be kept securely offline. The `truststore.jks` file should be published on a website somewhere along with the password for it. The remainder of the files will have to be loaded into the `nuts-discovery` app https://github.com/nuts-foundation/nuts-discovery

Vault docker image

An option to securely store the keys and distribute them is to let Hashicorp's Vault do the heavy lifting. The main idea is to mount the config/log/file dirs and then use the Vault command line tool to interact with Vault. When the container is offline, files from the *file* directory can be securely exchanged and the unseal keys can use a different medium for transfer.

To run the container:

```
docker run -it --cap-add=IPC_LOCK -v `pwd`/file:/vault/file -v `pwd`/log:/vault/log -
↪v `pwd`/config:/vault/config --name vault-nuts vault server
```

where *pwd* is the root of the structure used to exchange the vault store.

To interact with the container:

```
docker exec -it -e VAULT_ADDR=http://127.0.0.1:8200 vault-nuts /bin/sh
```

Initial setup

The initial setup for a new vault store:

```
vault operator init
```

This will generate an output like:

```
Unseal Key 1: 77EPX49RBoo2a8aN4+34XikzeqIhjvRVtHEQCCe6d7/l
Unseal Key 2: x4gJ52TgVHor6uA8ARuVZRyX228/8hAVLtzaVZnmpK9A
Unseal Key 3: z7lpYcr6So/tXCP2VEXPM88dIrxfPao0WUnYSmg9Hcl7
Unseal Key 4: whfbcUXM5ozcdB+21VwkhnSWvhui9eXF2ipefaYlrPRj
Unseal Key 5: O4EYtJOmgyiLY7g7gyp3Jq/QQ/DN99rUdnS8kkuLtlfv

Initial Root Token: s.4GhOLbGX0D3PsVxVV0p40Lea
```

The unseal keys have to be distributed amongst network operators.

Then we have to enable a key-value store, first unseal the store (3 times) using the unseal keys from above:

```
vault operator unseal
```

then:

```
export VAULT_TOKEN=s.4GhOLbGX0D3PsVxVV0p40Lea
vault secrets enable -path=nuts kv-v2
```

Storing/retrieving keys

With a root token (https://learn.hashicorp.com/vault/operations/ops-generate-root):

Enable the policy for accessing *nuts/*:

```
vault policy write secret /vault/config/secret-policy.hcl
```

Then create an access token:

```
vault token create -policy=secret
```

Use that token to store or get secrets:

```
export VAULT_TOKEN=s.zF801If9KeKnBYqBEP3vSTR1
vault kv put nuts/keys/root pem=s3cr3t
vault kv put nuts/keys/doorman pem=s3cr3t
vault kv put nuts/keys/network pem=s3cr3t
```

And read:

```
vault kv get nuts/keys/root
```

Closing

First destroy the root token:

```
export VAULT_TOKEN=s.4GhOLbGX0D3PsVxVV0p40Lea
vault token revoke s.4GhOLbGX0D3PsVxVV0p40Lea
```

Then close the docker container

Distribution

When using vault and after generating and storing the keys. The following file structure has to be distributed/backed-up:

- **root**
  - **keys**
    - doorman.crt
    - network_map.crt
    - root.crt
    - root.srl
    - truststore.jks
  - **vault**
    - **config**
      - default.hcl
      - secret-policy.hcl
    - file/**/*

CHAPTER 10

Installation

10.1 Nuts discovery service installation

Installation of the discovery service is as easy as running it from development (*Nuts discovery service development*). You can choose to create a runnable jar first

```
./gradlew bootJar
```

And then start it via

Make sure you use a java 8 compatible JVM.

10.2 Nuts consent cordapp installation

The corda part of the Nuts node (the registry and bridge are the two other parts) requires the following minimal file setup:

```
.
├── certificates
│   └── network-root-truststore.jks
├── cordapps
│   ├── contract-X.Y.Z.jar
│   └── flows-X.Y.Z.jar
├── corda-4.0.jar
└── node.conf
```

This structure will be extended when the node is run for the first time.

10.2.1 Truststore

The truststore must be obtained from Nuts or if you run a local network, you can copy it from *Nuts discovery configuration*.

10.2.2 Cordapps

The `contract` and `flows` jars are published on maven central. Obtain a copy: https://search.maven.org/search?q=nl.nuts or via browsing the repo: https://repo1.maven.org/maven2/nl/nuts/consent/cordapp/.

As an alternative you can use your own build, that will only work for a local network.

10.2.3 Corda.jar

Obtain a copy from here https://search.maven.org/search?q=a:corda or via browsing here: https://repo1.maven.org/maven2/net/corda/corda/4.0/.

Currently we're using version 4.0

10.2.4 Node.conf

This is the most interesting part of the setup. See *Nuts consent cordapp configuration*

10.2.5 First run

Before running, you can check the configuration with:

```
java -jar corda.jar --network-root-truststore cordacadevpass validate-configuration
```

The first time a node is started, it must be run with the additional `initial-registration` command:

```
java -jar corda.jar --network-root-truststore cordacadevpass initial-registration
```

This will generate the needed keys and apply for a certificate at the network authority.

10.2.6 Additional info

Additional info on the folder structure can be found on the corda documentation: https://docs.corda.net/node-structure.html and https://docs.corda.net/node-commandline.html

10.3 Using reverse proxies

The Nuts network makes heavy use of TLS connections with client certificates (a.k.a. mTLS). This documentation covers some guidelines and examples when using a reverse proxy for TLS termination and to secure your data endpoint and Nuts endpoints. It only covers HTTP/REST calls and not gRPC. In the *Examples* section, examples are given for the most popular proxies. This page only covers the client certificate part. Server certificates must be configured as normal, they must be signed by one of the globally accepted CAs (like Let's Encrypt and others).

10.3.1 Trusted certificates

> **Warning:** It is essential to do this part correctly, otherwise the endpoints will be exposed to the outside world!

Trusted roots

The basis for setting up security for mTLS is to get hold of the correct root certificates. For the internet, the trusted roots are preconfigured by the OS and/or browser. Root certificates are often published on websites. Check out https://letsencrypt.org/certificates/ for an example. For the Nuts network, several choices can be made, the matrix below shows a proposal on which CA bundle to trust.

Note: This is subject to change based on peer-reviews

	local	development	demo	test	production
nuts endpoints	none/local generated	Nuts dev CA	nuts demo CA	nuts test CA	nuts production CA
data endpoints	none	Nuts dev CA	nuts demo CA	PKIo with allow list	PKIo with allow list

Nuts endpoints cover all APIs provided by Nuts nodes. Data endpoints are provided by the vendor and cover patient related data. The **local**, **development** and **demo** CA bundles follow the same convention for both the nuts endpoints and the data endpoints. Since no real data will be used within these environments, using the different Nuts root certificates can simplify deployment.

For **test** and **production** this is a different story. **test** is a synonym for **acceptance**, thus real patient data is transferred. A national recognized trusted root in combination with specific trusted certificates gives maximum security. Configuring the different PKIo CAs in the reverse proxy will filter out most unwanted traffic and makes any traffic traceable to the responsible party. Using an allow list on top of that will give control to the vendor which certificates to accept. The allow list will be available from the registry. Only specifically published certificates get added to the allow list. More on the allow list can be read below. The different PKIo roots can not be obtained from the Nuts APIs.

Everywhere where the term *Nuts X CA* is used, the Nuts foundation acts as the **network authority**. The network authority role is only performed by a single party (legal entity) and it provides the CA bundles.

The Nuts foundation publishes its CA bundles on https://nuts.nl A full Nuts CA bundle would consist of a root, a Nuts intermediate and multiple vendor CAs.

The `/api/mtls/*` APIs on *Nuts registry API* describe how to obtain a list of CAs in various formats. Most reverse-proxies do not support calling APIs but only file based configuration. It's up to the vendor to convert the API output to a file and restart the proxy when something has changed. A single `truststore.pem` file with all CAs in PEM format is available in the directory configured by the `nuts.crypto.fspath` variable. This file can be synced or linked to from various configs.

Allow list

All client certificates used for mTLS will be published in the Nuts registry. This will allow for extracting the list of certificates that will be used on the application level. Vendors control which certificates from the allow list are passed on to the reverse-proxy, giving them complete control over which certificates get access. This means that vendors will have full control over which certificate is granted access and which isn't.

Most reverse-proxies require a configured list of CAs to enable mTLS. Most of the time, however, they will not be able to configure a list of accepted non-CA certificates. Specific certificate acceptance must therefore be done at the application level.

The `/api/mtls/*` APIs on *Nuts registry API* describe how to obtain an allow list in various formats.

10.3.2 Examples

Below are configuration examples on how to configure various reverse proxies.

Apache

Used apache version: 2.4.43

```
<VirtualHost *:443>
ServerName proxy-server
  # activate HTTPS server certificate on the reverse proxy
    SSLEngine on
    SSLCertificateFile "<Apache_home>/conf/proxy-server.crt"
    SSLCertificateKeyFile "<Apache_home>/conf/proxy-server.key"
  # activate the client certificate authentication
    SSLCACertificateFile "/etc/nuts/data/truststore.pem"
    SSLVerifyClient require
    SSLVerifyDepth 1
    SSLProxyEngine On
  # initialize the special headers to a blank value to avoid http header forgeries
    RequestHeader set SSL_CLIENT_CERT ""
    <Location /api/fhir>
        # forward certificate for allow list checking
        RequestHeader set SSL_CLIENT_CERT "%{SSL_CLIENT_CERT}s"
        ProxyPass http://localhost:8080
        ProxyPassReverse http://localhost:8080
    </Location>
</VirtualHost>
```

HAProxy

Note: help requested on a valid HAProxy example

Nginx

Used nginx version: 1.17.0

```
http {
  # only TLS > 1.2 is acceptable
  ssl_protocols TLSv1.2 TLSv1.3;
  ssl_prefer_server_ciphers on;

  access_log /var/log/nginx/access.log;
  error_log /var/log/nginx/error.log;

  # server on port 80 for HTTP -> HTTPS redirect
  server {
    listen 80;
    server_name example.com;
    return 301 https://example.com$request_uri;
  }
```

(continues on next page)

(continued from previous page)

```
  # The HTTPS server, which proxies our requests
  server {
    listen 443 ssl;
    server_name example.com;

    ssl_protocols TLSv1.2 TLSv1.3;
    # server certificate
    ssl_certificate /etc/nginx/ssl/example.com/fullchain.pem;
    ssl_certificate_key /etc/nginx/ssl/example.com/privkey.pem;

    # client certificate,
    # here we use the exported truststore.pem in the case Nuts is
    # running on the same machine
    ssl_trusted_certificate /etc/nuts/data/truststore.pem;
    ssl_verify_client on;

    access_log /var/log/nginx/example.com;

    location /api/fhir {
      proxy_set_header        Host $host;
      proxy_set_header        X-Real-IP $remote_addr;
      proxy_set_header        X-Forwarded-For $proxy_add_x_forwarded_for;
      proxy_set_header        X-Forwarded-Proto $scheme;
      # forward certificate for allow list checking
      proxy_set_header        X-Ssl-Client-Cert $ssl_client_cert;

      proxy_pass          http://localhost:8080;
    }
  }
}
```

Install software

CHAPTER 11

Configuration

11.1 Nuts consent bridge configuration

The *Nuts Consent Bridge* application is a Spring boot application. Therefore all Spring methods of configuring can be used including:

- Using a runtime JVM parameter specifying the spring configuration file: `java -jar myproject.jar --spring.config.location=/tmp/overrides.properties`
- Using environment variables, replacing all camelCasing and dots with underscores. So `nuts.consent.nats.address` becomes `NUTS_CONSENT_NATS_ADDRESS`

Property	Default	Description
nuts.consent.nats.address	nats://localhost:4222	The Nats address for events from and to *Nuts Service Space*.
nuts.consent.nats.cluster	test-cluster	The Nats clusterID.
nuts.consent.rpc.host	localhost	The host running the Consent Cordapp.
nuts.consent.rpc.port	7887	Port for Consent Cordapp.
nuts.consent.rpc.user	admin	Configured user on the RPC methods of the Consent Cordapp node.
nuts.consent.rpc.password	nuts	^^ same, but password ^^
nuts.consent.rpc.retryIntervalSeconds	5	Cooldown period before trying to reconnect to node.
nuts.consent.rpc.retryCount	0	How many times to reconnect (0 for infinite).
nuts.consent.registry.url	http://localhost:8088	The address + path where the Nuts registry is running.
nuts.consent.schedule.delay	15 * 60 * 1000	Delay between scheduled health checks in milliseconds.
nuts.consent.schedule.initial_delay	1000	Delay between scheduled health checks.
nuts.event.meta.location	.	Location where to store the timestamps of latest received Corda event

11.1.1 Docker

Nuts Consent Bridge doesn't have a DB but it does store a bunch of timestamps on disk. These timestamps are used to start listeners and receive messages/events beginning at the point in time when the application was closed. When using docker these files must be mounted otherwise each new version will consume all previous messages since the beginning of time (epoch = 0).

The default location for *nuts.event.meta.location* == .. This has to be changed to something else. An example config would be:

```
nuts.event.meta.location: /var/nuts/timestamps
```

The docker run command would then be:

```
docker run \
  --name bridge \
  -w /opt/nuts \
  -v {{config_dir}}/application.properties:/opt/nuts/application.properties \
  -v {{data_dir}}:/var/nuts/timestamps \
  -p 8080:8080 \
  -d \
  nuts-consent-bridge:latest-dev
```

So besides the *application.properties*, a data dir has to be mounted as well.

11.2 Nuts discovery configuration

Before the *Nuts Discovery Service* can be started a few keys and certificates need to be generated. All OpenSSL commands use config files for the correct generation of certificates and keys. Windows scripts are currently lacking.

By default it'll try to find the following keys at the given location. All files are in PEM format

These locations can be overriden by providing an alternative properties file with the following contents

```
nuts.discovery.cordaRootCertPath = keys/root.crt
nuts.discovery.intermediateKeyPath = keys/doorman.key
nuts.discovery.intermediateCertPath = keys/doorman.crt
nuts.discovery.networkMapCertPath = keys/network_map.crt
nuts.discovery.networkMapKeyPath = keys/network_map.key
nuts.discovery.nutsRootCertPath = keys/nuts_root.crt
nuts.discovery.nutsCAKeyPath = keys/nuts_ca.key
nuts.discovery.nutsCACertPath = keys/nuts_ca.crt
nuts.discovery.certificateValidityInDays = 1095
nuts.discovery.contractHashes =␣
↪6ACDE387C0DF227A6C4ED77407B58E9103C2EA1A66796CE37BC497931F4E1631
nuts.discovery.flowHashes =␣
↪5f60201e5f4e698300f3baf94dad1517a1314b4f406fd90610a78d798ffe972d
nuts.discovery.autoAck = true
```

The alternative config file can be passed to the executable by param like this

```
java -jar nuts-discovery.jar --spring.config.location=file:./custom.properties
```

Individual properties can also be overriden by passing them via the command-line

```
java -jar nuts-discovery.jar --nuts.discovery.networkMapKeyPath=keys/network_map.key
```

Or by using environment variables

```
NUTS_DISCOVERY_NETWORK_MAP_KEY_PATH=keys/network_map.key java -jar nuts-discovery.jar
```

Besides the keys and certificates it's also possible to change the `server.port` property.

11.2.1 Key generation

Generate root key and certificate

Run the `generate_keys.sh` script to create a `keys` folder with all the needed keys and certificates.

```
./generate_keys.sh
```

11.2.2 Deployment with Helm

Installation

In the following examples we use the *development* namespace. The values.yaml currently contains values for development.

```
helm install --debug --name discovery --namespace development charts/nuts-discovery -
↪f charts/nuts-discovery/values.yaml
```

Upgrading

```
helm upgrade discovery -f charts/nuts-discovery/values.yaml charts/nuts-discovery --
↪namespace development --recreate-pods
```

11.3 Nuts Network configuration

11.3.1 SSL/TLS Deployment Layouts

This section describes which deployment layouts are supported regarding SSL/TLS. In all layouts there should be a valid X.509 server certificate and private key issued by a publicly trusted Certificate Authority if the node operator wants other Nuts nodes to be able to connect to the node.

Direct WAN Connection

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 present on the Nuts node and configured in the Nuts Network engine.

SSL/TLS Offloading

In this layout incoming TLS traffic is decrypted on a SSL/TLS terminator and then being forwarded to the Nuts node. This is 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 certificates presented to the SSL/TLS terminator must be forwarded as a header to the Nuts node.
- SSL/TLS terminator must use the truststore managed by the Nuts node (which is sourced from the Nuts Registry) as root CA trust bundle.

SSL/TLS Pass-through

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.

11.4 Nuts consent cordapp configuration

The basic node.conf inside the Cordap base directory should look similar like this:

```
myLegalName="O=Nuts,C=NL,L=Groenlo,CN=nuts_corda_development"
emailAddress="info@nuts.nl"
devMode=false
devModeOptions {
  allowCompatibilityZone = true
}
networkServices {
    doormanURL = "http://localhost:8080"
    networkMapURL = "http://localhost:8080"
}
p2pAddress="localhost:17886"
rpcSettings {
    address="localhost:11003"
    adminAddress="localhost:11043"
}
rpcUsers=[]
custom = {
    jvmArgs: [ "-Xmx1G", "-XX:+UseG1GC" ]
}
```

Both the `doormanURL` and `networkMapURL` must point to the location where *Nuts Discovery* is running. The `p2pAddress` is the endpoint that must be exposed to the outside world and which is added to the *Nuts registry*. The `rpcSettings` property is used for exposing the rpc endoint used by *Nuts consent bridge*.

The `myLegalName` is the identity of the node and must be unique. It follows the x500 name convention. This is also the identiy that is added to the *Nuts registry* consent endpoint.

Since Corda 4.4 memory consumption has changed, the default 512m is no longer enough. The *custom* section is therefore mandatory:

```
custom = {
    jvmArgs: [ "-Xmx1G", "-XX:+UseG1GC" ]
}
```

11.4.1 Database & Docker

By default Corda places the DB in the *baseDirectory* which, by default, is inside a docker container. This can be avoided by mounting the entire *baseDirectory* but this also means the cordapps and *corda.jar* have to be mounted as well. The Nuts cordapp image has these inside the image. Having to download them again is extra work, that's just annoying. Luckily it's also possible to put the DB in a different location. The default DB configuration is below:

```
dataSourceProperties = {
    dataSourceClassName = org.h2.jdbcx.JdbcDataSource
    dataSource.url = "jdbc:h2:file:"${baseDirectory}"/persistence;DB_CLOSE_ON_
↪EXIT=FALSE;WRITE_DELAY=0;LOCK_TIMEOUT=10000"
    dataSource.user = sa
    dataSource.password = ""
}
```

By putting the DB in a sub directory it'll be easier to mount. For example changing above config to:

```
dataSourceProperties = {
    dataSourceClassName = org.h2.jdbcx.JdbcDataSource
    dataSource.url = "jdbc:h2:file:"${baseDirectory}"/data/persistence;DB_CLOSE_ON_
↪EXIT=FALSE;WRITE_DELAY=0;LOCK_TIMEOUT=10000"
    dataSource.user = sa
    dataSource.password = ""
}
```

places the DB in a */data* subdirectory. Which can then be mounted with:

```
docker run \
    -v {{data_dir}}:/opt/nuts/data \
    -d \
    nuts-consent-cordapp:latest-dev
```

11.4.2 Signed libraries

When `devMode=false` Corda requires signed or whitelisted jars containing the digital contracts. At this point it's undecided if Nuts is going to start with whitelisted jars or is it going to start with signed jars. When running with `devMode=true` this is of no concern.

11.4.3 Additional info

See https://docs.corda.net/corda-configuration-file.html

11.5 Nuts service config

The Nuts-go library contains some configuration logic which allows for usage of configFiles, Environment variables and commandLine params transparently. If a Nuts engine is added as Engine it'll automatically work for the given engine. It is also possible for an engine to add the capabilities on a standalone basis. This allows for testing from within a repo.

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`

11.5.1 Options

The following options can be configured:

Key	Default	Des
address	localhost:1323	Add
configfile	nuts.yaml	Nut
identity		Ven
mode	server	Mo
strictmode	false	Wh
verbosity	info	Log
Auth		
auth.actingPartyCn		The
auth.address	localhost:1323	Inte
auth.enableCORS	false	Set
auth.irmaConfigPath		path
auth.irmaSchemeManager	pbdf	The
auth.mode		serv
auth.publicUrl		Pub
auth.skipAutoUpdateIrmaSchemas	false	set i
ConsentBridgeClient		
cbridge.address	http://localhost:8080	API
ConsentStore		
cstore.address	localhost:1323	Add
cstore.connectionstring	:memory:	Db
cstore.mode		serv
Crypto		
crypto.fspath	./	whe
crypto.keysize	2048	num
crypto.storage	fs	stor
Events octopus		
events.autoRecover	false	Rep
events.connectionstring	file::memory:?cache=shared	db c
events.incrementalBackoff	8	Incr
events.maxRetryCount	5	Max
events.natsPort	4222	Port

Table

Key	Default	Des
events.purgeCompleted	false	Purg
events.retryInterval	60	Retr
Network		
network.address		Inte
network.bootstrapNodes		Spa
network.certFile		PEM
network.certKeyFile		PEM
network.grpcAddr	:5555	Loc
network.mode		serv
network.nodeID		Inst
network.publicAddr		Pub
network.storageConnectionString	file:network.db	SQL
Registry		
registry.address	localhost:1323	Inte
registry.clientTimeout	10	Tim
registry.datadir	./data	Loc
registry.mode	server	serv
registry.organisationCertificateValidity	365	Nur
registry.syncAddress	https://codeload.github.com/nuts-foundation/nuts-registry-development/tar.gz/master	The
registry.syncInterval	30	The
registry.syncMode	fs	The
registry.vendorCACertificateValidity	1095	Nur
Validation		
fhir.schemapath		loca

Key	Default	Des
address	localhost:1323	Add
configfile	nuts.yaml	Nut
identity		Ven
mode	server	Mod
strictmode	false	Wh
verbosity	info	Log
Auth		
auth.actingPartyCn		The
auth.address	localhost:1323	Inte
auth.enableCORS	false	Set
auth.irmaConfigPath		path
auth.irmaSchemeManager	pbdf	The
auth.mode		serv
auth.publicUrl		Pub
auth.skipAutoUpdateIrmaSchemas	false	set i
ConsentBridgeClient		
cbridge.address	http://localhost:8080	API
ConsentStore		
cstore.address	localhost:1323	Add
cstore.connectionstring	:memory:	Db
cstore.mode		serv

Table

Key	Default	Des
Crypto		
crypto.fspath	./	whe
crypto.keysize	2048	num
crypto.storage	fs	stor
Events octopus		
events.autoRecover	false	Rep
events.connectionstring	file::memory:?cache=shared	db c
events.incrementalBackoff	8	Incr
events.maxRetryCount	5	Max
events.natsPort	4222	Port
events.purgeCompleted	false	Purg
events.retryInterval	60	Retr
Network		
network.address		Inte
network.bootstrapNodes		Spa
network.certFile		PEM
network.certKeyFile		PEM
network.grpcAddr	:5555	Loc
network.mode		serv
network.nodeID		Inst
network.publicAddr		Pub
network.storageConnectionString	file:network.db	SQL
Registry		
registry.address	localhost:1323	Inte
registry.clientTimeout	10	Tim
registry.datadir	./data	Loc
registry.mode	server	serv
registry.organisationCertificateValidity	365	Nur
registry.syncAddress	https://codeload.github.com/nuts-foundation/nuts-registry-development/tar.gz/master	The
registry.syncInterval	30	The
registry.syncMode	fs	The
registry.vendorCACertificateValidity	1095	Nur
Validation		
fhir.schemapath		loca

11.6 Nuts registry configuration

11.6.1 Sync modes

The registry supports two modes for updating the internal Db: a file system watcher (`fs`) or downloading from Github (`github`). When using Github, the registry checks every `syncInterval` minutes if anything has changed on Github. The `syncAddress` must point to a tar.gz with the needed registry files included. Github has a nice URL for this. By default it uses the config in the master branch.

what to tweak

CHAPTER 12

Administration

12.1 Nuts Crypto Administration

This administration guide will help you to achieve the following goals:

- generate-vendorca-csr which is required for registering your vendor.
- *2. Self-signing vendor CA certificate* which is required for registering your vendor when there is no central Network Authority on the network.

12.1.1 1. Generating vendor CA certificate CSR

This command generates a vendor CA certificate CSR for the current vendor. The resulting CSR should then be signed (in other words, a certificate is issued) by the Network Authority. Upon receiving the certificate it is used to register the vendor.

Note: If your network does not have a central Network Authority vendor should self-sign the certificate. See *2. Self-signing vendor CA certificate*.

If there is a key pair in the crypto module for the vendor it will be associated with the CSR (and thus the issued certificate). It will be generated if it doesn't exist.

To generate the CSR you need your vendor's name as it will end up as *Subject* in the CSR.

The syntax of this command is as follows:

```
./nuts crypto generate-vendor-csr <name>
```

To generate a CSR for vendor "BecauseWeCare B.V.", run the following command:

```
NUTS_MODE=cli ./nuts crypto generate-vendor-csr "BecauseWeCare B.V."
```

If the command completes successfully, it outputs the CSR (as PEM-encoded PKCS#10). This CSR should then be sent to the Network Authority. Upon receiving the issued certificate proceed with *1. Registering a vendor*.

12.1.2 2. Self-signing vendor CA certificate

This command self-signs a CA certificate for the current vendor, to be used when there's no Network Authority. The resulting certificate is used to register the vendor.

If there is a key pair in the crypto module for the vendor it will be associated with the certificate. It will be generated if it doesn't exist.

To self-sign the certificate you need your vendor's name as it will end up as *Subject* and *Issuer* in the certificate.

The syntax of this command is as follows:

```
./nuts crypto selfsign-vendor-cert <name>
```

To self-sign a certificate for vendor "BecauseWeCare B.V.", run the following command:

```
NUTS_MODE=cli ./nuts crypto selfsign-vendor-cert "BecauseWeCare B.V."
```

If the command completes successfully, it outputs the certificate (PEM-encoded). This certificate can be used for *1. Registering a vendor*.

12.2 Nuts Registry Administration

This administration guide will help you to achieve the following goals:

- *1. Registering a vendor* (representing your company).
- *2. Registering a care organization* as vendor customer, which allows you to register endpoints.
- *3. Registering an endpoint* for the care organization.
- *4. Updating an existing endpoint* of the care organization.
- *5. Verifying and fixing registry data*.
- *6. Refreshing vendor CA certificate* of your registered vendor.
- *7. Refreshing organization certificate* of one of your vendor's organizations.

12.2.1 Updating the central Nuts Registry

Currently, the data of the Nuts Registry is centrally hosted on Github. When you want other Nuts nodes to know about your vendor, care organizations and endpoints you have to submit the data, produced by the administration commands, to get it included in the registry. To do so, you have to create a pull request on Github for the nuts-foundation/nuts-registry-development repository. Should you be unable to do so, contact the Nuts Foundation for alternatives.

The event data generated by the commands (which you should submit to the central registry) can be copied from either the CLI console or by copying the files from the registry storage on disk.

Copying files from disk

If you have access to the file system where your Nuts node stores its data, you can simply copy the event files from there. They are typically stored in the *data/events* directory.

Copying from CLI console

Alternatively you can copy the output of the CLI commands, since they output the event file name and payload, e.g.:

```
Event: 20200224123533008-RegisterVendorEvent.json
{"type":"RegisterVendorEvent","issuedAt":"2020-02-24T13:35:33.008076348+01:00",
↪"payload":{"identifier":"urn:oid:1.3.6.1.4.1.54851.4:00000001","name":
↪"BecauseWeCare B.V."}}
```

Note: In the (near) future we will move away from Github to a decentralized registry.

12.2.2 1. Registering a vendor

This command registers a vendor in the registry, giving it further access to the Nuts Registry. Afterwards, care organizations can be registered as the organization's clients. When a vendor is registered, a CA certificate is issued. This certificate is used (by the vendor) to issue certificates, e.g. care organization certificates.

To register a vendor, you need its name (as registered at the Chamber of Commerce) and its identifier should be configured as the node's `identity`, which is the company's URN-encoded Chamber of Commerce registration number.

The syntax of this command is as follows (parameter 'domain' defaults to 'healthcare'):

```
./nuts registry register-vendor <name> <domain>
```

To register vendor "BecauseWeCare B.V." identified by Chamber of Commerce registration number "00000001", run the following command ('domain' defaults to 'healthcare'):

```
NUTS_MODE=cli ./nuts registry register-vendor "BecauseWeCare B.V."
```

If the command completes successfully, it should output the message: "Vendor registered."

Note: The vendor CA certificate is currently self-signed. In the future, the vendor CA certificate will be issued by the Nuts Foundation.

12.2.3 2. Registering a care organization

When a vendor is known in the Nuts Registry, it can register organization by claiming them (as client). Afterwards the vendor can register endpoints for the organization which are served by the vendor's Nuts node. When an organization is registered, the vendor CA issues an organization certificate. Its key is used for encrypting data exchanges and signing registry operations (e.g. registering endpoints).

To register an organization you need the organization's name and its identifier, which is the organization's URN-encoded AGB-code.

The syntax of this command is as follows:

```
./nuts registry vendor-claim <organization-identifier> <organization-name>
```

For example:

```
NUTS_MODE=cli ./nuts registry vendor-claim urn:oid:2.16.840.1.113883.2.4.6.1:123456
↪"Kunstgebit Thuiszorg"
```

If the command completes successfully, it should output the message: "Vendor organization claim registered"

Note: Registering an organization as vendor client is called *claiming* because in future instead of the vendor solely registering an organization being its client, the organization has to do the same (claim being a client of a software vendor). Only if both entities claim to have a relationship with each other, the organization is registered being a client of the vendor.

12.2.4 3. Registering an endpoint

After registering an organization, the vendor can administer its endpoints. The endpoints are used by other Nuts nodes when they want to exchange data with the Nuts node serving a particular organization.

The syntax of this command is as follows:

```
./nuts registry register-endpoint <organization-identifier> <type> <url>
```

In the following example we register a Corda consent endpoint for the previously registered organization:

```
NUTS_MODE=cli ./nuts registry register-endpoint urn:oid:2.16.840.1.113883.2.4.6.
↪1:123456 \
    urn:nuts:endpoint:consent \
    "tcp://1.2.3.4:4321" \
    -i "urn:ietf:rfc:1779:O=Kunstgebit Thuiszorg,C=NL,L=Franeker,
↪CN=kunstgebitthuiszorg_nuts_cordapp_development"
```

Don't forget to replace the ID flag (`-i`) with the correct subject DN from the node's X.509 certificate (which is specific for this endpoint type).

Note: Endpoint ID is application specific, some endpoint types (e.g. `urn:nuts:endpoint:consent`) require a specific ID for others it doesn't matter and the randomly generated ID is fine. When registering an endpoint for a Bolt, please refer to the Bolt documentation for any specifics.

In addition the following flags can be supplied:

Flag	Description	Example
-i	Identifier for the endpoint. If not supplied a type 4 UUID is randomly generated.	*-i abc*
-p	Endpoint metadata in the form of string properties, specified as **key=value**	*-p foo=bar*

12.2.5 4. Updating an existing endpoint

To update an endpoint, simply register it again using the `register-endpoint` command using the same ID. The update completely replaces the previous registration, so specify all relevant fields and properties. Don't forget to specify the ID (using the `-i` flag) if it was auto-generated during endpoint registration.

12.2.6 5. Verifying and fixing registry data

In certain circumstances the registry data owned by your node might need maintenance. Examples are when certificates are missing or close to expiry, records that need to be migrated to a newer version (format) or missing signatures. The Nuts node performs verification on startup and will inform the you (through the logs) of any issues that need to be fixed.

To verify the data using the CLI use the `verify` command:

```
NUTS_MODE=cli ./nuts registry verify
```

If your node's registry data needs fixing this command will inform you:

```
Verification complete, data must be fixed. Please rerun command with --fix or -f
```

To apply the fixes, rerun the command with the `-f` flag:

```
NUTS_MODE=cli ./nuts registry verify -f
```

When data is fixed new events are emitted. It's **very important** that these events are submitted to the central Nuts registry (please refer to *Updating the central Nuts Registry*).

Note: It's recommended to verify (and fix if necessary) your node's data after each upgrade to ensure compatibility in the long term. The best way to spot problems is to inspect your node's logs when starting it after upgrading.

12.2.7 6. Refreshing vendor CA certificate

At some point the vendor CA certificate must be refreshed (issued again). Most often because the current certificate is nearing the end of its validity. If the vendor doesn't have a certificate yet this command is used to issue it. This command will **not** rotate (create a new) the key pair for the vendor but will create one if it doesn't exist.

The syntax of this command is as follows:

```
./nuts registry refresh-vendor-cert
```

12.2.8 7. Refreshing organization certificate

This works exactly the same as *6. Refreshing vendor CA certificate* but works on an organization. The syntax of this command is as follows:

```
./nuts registry refresh-organization-cert <organization-identifier>
```

In the following example we refresh the certificate for a previously registered organization:

```
NUTS_MODE=cli ./nuts registry refresh-organization-cert urn:oid:2.16.840.1.113883.2.4.
↪6.1:123456
```

This section describes how to administer your Nuts instance in a production environment. It assumes you have a running Nuts node. If you don't have a running Nuts node yet, please refer to the *Installation* guide.

CHAPTER 13

Monitoring

13.1 Nuts consent bridge monitoring

13.1.1 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 with a 200 status code.

13.1.2 Basic diagnostics

```
GET /status/diagnostics
```

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

```
General status: DOWN
nutsEventListener=UP {}
nutsEventPublisher=UP {}
&cordaRPCClientFactory=DOWN {}
cordaNotaryPing=UP {}
cordaRandomPing=UP {}
```

13.1.3 Extensive diagnostics

The bridge uses Spring Actuator for health monitoring. A basic overview can be viewed at:

```
GET /actuator/health
```

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

```
{"status":"DOWN","details":{"nutsEventListener":{"status":"UP"},"nutsEventPublisher":{
↪"status":"UP"},"&cordaRPCClientFactory":{"status":"DOWN"}}}
```

13.1.4 Docker monitoring

A docker *HEALTHCHECK* is available on the image, it runs *curl localhost:8080/status*. This can be connected to your favourite monitoring software. Output can be checked by using *docker inspect –format='{{json .State.Health}}' container*

13.2 Nuts consent cordapp monitoring

The Corda node in itself offers quite some monitoring options. See: https://docs.corda.net/node-administration.html#monitoring-your-node.

13.2.1 Docker monitoring

A docker *HEALTHCHECK* is available on the image, it runs *netstat -an | grep 7886*. This can be connected to your favourite monitoring software. Output can be checked by using *docker inspect –format='{{json .State.Health}}' container*

13.3 Nuts certificate monitoring

One of the challenges with using certificates and key management is to renew them in time. The first step in the process is to identify which certificates are about to expire and alert an administrator about this fact. Therefore some prometheus metrics are provided which give the amount of certificates about to expire within a certain period.

13.3.1 Prometheus metrics

```
# HELP nuts_crypto_certificate_expiry a gauge on the amount of certificates about to␣
↪expire.
# TYPE nuts_crypto_certificate_expiry gauge
nuts_crypto_certificate_expiry{period="day"} 0
nuts_crypto_certificate_expiry{period="week"} 0
nuts_crypto_certificate_expiry{period="4_weeks"} 1
```

13.4 Nuts service executable monitoring

13.4.1 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.

13.4.2 Basic diagnostics

```
GET /status/diagnostics
```

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

```
Status
    Registered engines: Status,Logging,Crypto,Registry,Events octopus,
↪ConsentLogicInstance,ConsentStore,Validation,Auth,ConsentBridgeClient
Logging
    verbosity:
Events octopus
    Nats streaming server: mode: STANDALONE @ 0.0.0.0:4222, ID: nuts, last error: NONE
    DB: ping: true
```

13.4.3 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`

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'
```

(continues on next page)

(continued from previous page)

```
    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!

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
# 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 table.
# 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␣
↪used by the GC since the program started.
# 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␣
↪metadata.
# 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
```

(continues on next page)

(continued from previous page)

```
# 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␣
↪collection.
# 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
# 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.
```

(continues on next page)

(continued from previous page)

```
# 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
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
```

Some introduction on how to monitor the services

CHAPTER 14

Production Considerations

This section describes what to consider when setting up your Nuts node in a production environment.

14.1 Monitoring

Make sure you monitor the indiviual Nuts services (*Monitoring*).

CHAPTER 15

Migration

This chapter describes which migrations are performed when upgrading your Nuts node. Most of them will be automatic, but sometimes manual action is required.

15.1 0.15

Starting 0.15 the node will default to vendor CA certificates issued by the Nuts Network Authority instead of self-signing them. See *1. Registering a vendor* how to register (or update) your vendor. There is backwards compatibility by supporting self-signed certificates issued before version 0.15.

In 0.15 the Nuts Network is introduced, an experimental P2P network decentralized document transport layer which is intended as a replacement candidate for Nuts Registry data on Github and Consents over Corda.

Manual action required

While the Nuts Network engine is enabled by default it needs configuration to function optimally:

1. Configure bootstrap node(s) using which peer nodes can be discovered; since there's no dedicated bootstrap node (at time of writing) it has to be configured with the Nuts Network address of one (or more) of the node's peer vendor nodes. This is done using the *NUTS_NETWORK_BOOTSTRAPNODES* which takes a space-separated list of nodes (e.g. *vendor-b:5555 vendor-c:9876*) to initially connect to.
2. Configure *NUTS_NETWORK_GRPCADDR* which is the local interface address gRPC (which powers Nuts Network) will bind on. Defaults to *:5555* but should be changed if applicable. This interface should be publicly reachable (but can be behind a TLS forwarding proxy). TLS termination for incoming connections using a proxy in front of the node is currently not possible.
3. Configure *NUTS_NETWORK_PUBLICADDR* which is the network address (*host:port*) which will be advertised on the network so that your node can be discovered by other nodes you're not directly connected to. This will typically be used to support resolving by public DNS entries (e.g. *nuts.vendor-A.nl:5555*), when the Nuts node is behind a proxy/load balancer or NAT.

Refer to *Nuts Network configuration* for configuration guidelines for your particular infrastructure layout.

15.2 0.14

Starting version 0.14 vendors and organizations will have an X.509 certificate (encoded in the JWK) associated with their key pairs. These certificates are used to identify the holder of the key pair used to sign events, which is also introduced in this version.

Manual action required: This migration can be performed by the Nuts node; use the newly introduced *registry verify* command with the *–fix* flag to generate key pairs (if necessary), issue certificates and sign events. Don't forget to publish these changes to the central registry.

15.3 0.13

In 0.13 the Node *identity* configuration parameter is introduced, which identifies the vendor operating the Node. It is used to determine which registry entries it owns an should manage.

Manual action required: While this is a mandatory parameter, operators should configure it in either nuts.yaml or through environment variables. To learn how to configure this parameter please refer to *Nuts service config*.

CHAPTER 16

Contact

16.1 Information

More information about the Nuts foundation can be found at nuts.nl

16.2 Communication

The main means of communication is via Slack.

HTTP Routing Table

/(request_id)

/certificate

/network-map