

Introduction

OYDID (Own Your Decentralized IDentifier) provides an open, self-sustained, trustless environment for managing Decentralized Identifiers (DIDs). The oyd:did method links the identifier cryptographically to the DID Document and through also cryptographically linked provenance information in a public log it ensures resolving to the latest valid version of the DID Document.

DID Document structure

```
{"doc":{"JSON Object"},"key":"public keys","log":"pointer to log entry and location"}
```

DID is the base58 encoded sha256 hash of the DID Document and can include a default location to retrieve the associated DID Document (appended and separated with ";")

- doc: publicly accessible and up-to-date information
- key: public document and revocation key to be used in verifying the log (using Ed25519 described in [RFC 8032](#), base58 encoded and separated with ":",")
- log: pointer to provenance information of DID document (hash of Terminate log entry)

Log structure - an array with each item associated to a DID:

```
{"ts":int,"op":int,"doc":"hash","sig":"signature","previous":[array]}
```

Timestamp (ts POSIX epoch), other items are used based on Operation (op):

- Terminate (op=0): confirms last entry until revoke entry is published
doc: hash of revoke entry, sig: doc signed by private document key
previous: can reference Clone or Delegate log entries for confirmation
- Revoke (op=1): invalidates a Terminate log entry; only published for new information
doc: hash of doc and key in DID Document, sig: doc signed by private revocation key
previous: can reference Create or Update, and always Terminate
- Create (op=2): start new DID
doc: hash of DID Document, sig: doc signed by private document key
previous: has only a reference when created using clone pointing to this clone entry
- Update (op=3): update DID
doc: hash of DID Document, sig: doc signed by private document key
previous: reference previous revoke log entry
- Clone (op=4): create linked DID with same "doc" but new key and log (see figure)
doc: hash of new DID Document, sig: doc signed by the new private document key
previous: reference Create or Update log entry
- Delegate (op=5): publish public document and/or revocation key for delegation
doc: public document and/or revocation key, sig: doc signed by private document key
previous: reference Create, Update, or Terminate log entry (active from this ts)
- Challenge (op=6): mark an untrusted revocation or delegation
doc: hash of revoke/delegate entry, sig: doc signed by master private revocation key
previous: references Revoke/Delegate log entry that is challenged

The log can be interpreted as a directed acyclic graph and represents the life cycle of a DID.

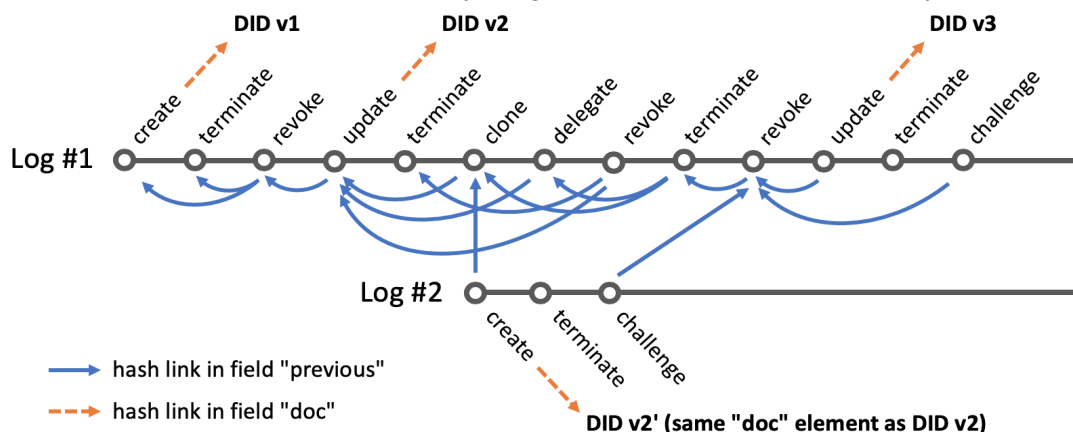


Figure 1: DAG with DID resolving to v2

OYDID USPs

The following properties are different in the did:oyd method compared to other existing methods¹:

- content-based addressing: does not rely on any black-box components to map a DID to the DID document but provides pure cryptographic provenance from the DID Document to the DID; it is based on only 3 operations: signatures (signed by private key and verifiable with public key), hashing (using SHA-256), and binary-to-text encoding (using Base58)
- local: run all components locally or on your own servers while still being decentralized through referencing other locations (using clone)
- low cost: independent of a 3rd party storage system or DLT
- simple: 1-page specification

Appendix A: Scenarios & Examples for standalone oydid usage

Alice (A) shares data with Bob (B) and delegates updates to Dave (D); Eve (E) tries to interfere.

Prerequisites:

- get oydid command from <https://github.com/OwnYourData/did-cmd/>
- oydid/did-base image to host own DIDs is available at <https://hub.docker.com/r/oydid/did-base> (Swagger: <https://api-docs.ownyourdata.eu/oydid/>)

1) A creates DID to document available service endpoint:

```
echo '{"service":"https://business.data-container.net/api/data"}' | oydid create
```

- start in empty directory
- store DID Document and Log at default location <https://oydid.ownyourdata.eu>
- private keys for document and revocation signatures stored in local directory (*key.b58)
- revocation document stored in local directory (*revocation.json)
- print DID to stdout (did:oyd:123aBz)
- show log information with command
oydid log did:oyd:123aBz

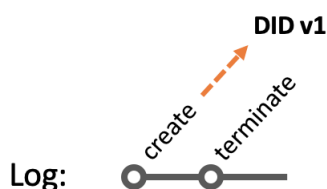


Figure 2: simple log sequence with only 2 entries after creating DID

2) B reads DID:

```
oydid read 123aBz
```

- print DID document
- Hint: don't forget apostrophes for non-default locations or otherwise the location is stripped
- example: `oydid read "123aBz;https://oydid.ownyourdata.eu"`

Cryptographic proof about link between DID and DID Document, ownership, and completeness is performed in the following steps:

- a) the identifier is the SHA256 hash value of the DID Document
- b) the DID Document includes public keys and the SHA256 hash value of a DID log entry
`{"doc":{"service":"..."}, "key":"123:abc", "log":"789xYz"}`

¹ <https://www.w3.org/TR/did-spec-registries/#did-methods>

- c) the log entry (create) for the DID Document provides a signature of the identifier to prove possession of the private key
 {"ts":123, "op":2, "doc":"123aBz","sig":"...", "previous":[]}
- d) the other log entry (terminate) provides revocation info; in case a log entry with the stated hash in doc exists the DID resolving process must continue
 {"ts":124, "op":0, "doc":"059mNr","sig":"...", "previous":[]}

3) A updates DID Document:

```
echo '{"service":"https://biz2.data-container.net/api/data"}' | oydid update 123aBz
```

- require files created from 1) in working directory (keys and revocation info)
 - update DID Document and Log at <https://oydid.ownyourdata.eu>
 - revoke previous Terminate log entry by publishing revocation document in Log
 - new revocation document stored in working directory
 - print new DID to stdout (did:oyd:456aBz)
 - B retrieves new document with old DID:
 oydid read hash 123aBz
 - show log information with command
 oydid log 123aBz
- Hint: you can omit the prefix did:oyd and it does not matter if you use the identifier from v1 (123aBz) or v2 (456aBz) as those are equivalent

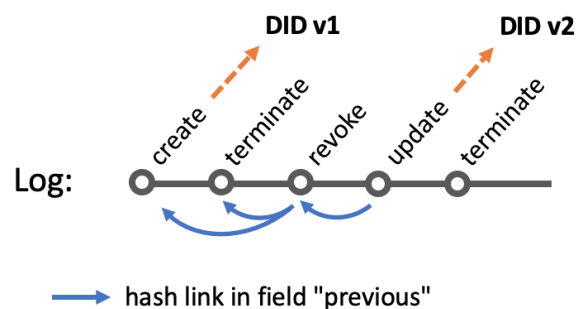


Figure 3: log sequence after updating DID

4) A deactivates DID

```
oydid revoke 456aBz
```

- require files created from 1) in working directory (keys and revocation info)
- update Log at <https://oydid.ownyourdata.eu> by publishing revocation info
- trying to resolve DID will produce error
- to show complete log history use logs operation (note additional "s")
 oydid logs 456aBz

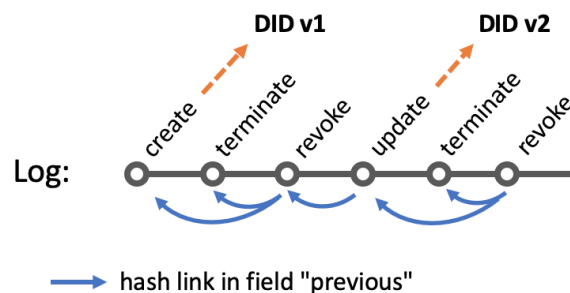


Figure 4: log sequence of deactivated DID

5) B clones A's DID Document:

allows B to maintain a copy of A's DID in case A's hosting is not available

```
oydid clone 456aBz -l https://did2.data-container.net
```

- make sure to not use the deactivated DID from step 4 and require files created from 1) in working directory (keys and revocation info)
 - create linked DID Document and Log at <https://did2.data-container.net>
 - add log entry at default location <https://oydid.ownyourdata.eu>
 - same document as 456aBz but with different keys and own Log
 - private keys for document and revocation signatures stored in working directory
 - revocation document stored in working directory
 - print DID (oyd:did:789aBz; <https://did2.data-container.net>)
 - note that B's clone is not yet active since the Terminate from 3) is still valid - see 7)

6) A gives D delegation rights:

allows D to make updates to the DID document and Log while using its own private keys

```
echo '{"key":"D's public doc key:D's public revocation key"}' | oydidd delegate 456aBz
```

- requires A's document private key in working directory for signing
 - creates a new log entry with doc holding key information
 - note that this delegation is not yet active since the Terminate from 3) is still valid - see 7)

7) A confirms B's clone and D's delegation rights:

mark log entries after last Terminate as valid by issuing a new Terminate

```
echo '["log hash #1", "log hash #2"]' | oydidd confirm oyd:did:456aBz
```

- require keys and revocation information created from 3) in working directory
- hash values for log entries can be shown with

```
oydid log oyd:did:456aBz --show-hash
```

 - update Log at default location <https://oydid.ownyourdata.eu> by publishing the Revocation entry and creating a new Termination entry referencing the input array as "previous"

8) E steals A's keys and creates update:

demonstrate recovery strategy when the DID owners keys are compromised and wrong information is published

- E publishes revocation log entry, updates DID Document and adds new termination
 - A and B (holder of a confirmed clone) publish challenges against wrongful revocation

```
A: oydidd challenge revocation-log-hash;https://oydid.ownyourdata.eu
B: oydidd challenge revocation-log-hash;https://oydid.ownyourdata.eu \
-l https://did2.data-container.net
```
 - entries are stored in the respective logs and resolving the DID omits any entries after E's revocation
 - A creates new keys and publishes an update following the last valid termination entry; in case E challenges this update (because of private key access) B needs to clone the new update with the same keys as in 4) (acting as confirmation) leading to 2 confirmation and only 1 challenge (in this consensus finding only clones are allowed from users predating the first challenged termination)

9) E gains access to A's hosting and prohibits updates

- E deletes every new log entry that would update the DID Document
- A publishes updates on validated clone log using keys from 3), i.e., takes over cloned copy

Appendix B: Examples for using oydid with Semantic Containers

This is a simple walk-through for starting a container, assigning a DID and then assigning DIDs to individual records.

Prerequisites

- get the latest Semantic Container base image
docker pull semcon/sc-base:latest
- have oydid command line tool installed:
run install.sh from here: <https://github.com/OwnYourData/did-cmd/blob/main/install.sh>
and make sure to include ~/bin in your path:
export PATH="\$PATH:\$HOME/bin"
- other tools used in the examples are curl and jq

1) start a Semantic Container

```
export CONTAINER_NAME=demo
```

```
export SEMCON_URL="http://localhost:4000"
```

```
IMAGE=semcon/sc-base:latest; docker run -d --name $CONTAINER_NAME -p 4000:3000 \  
-e AUTH=true \  
-e IMAGE_SHA256="$(docker image ls --no-trunc -q $IMAGE | tail -1)" \  
-e IMAGE_NAME=$IMAGE \  
-e SERVICE_ENDPOINT="http://1.2.3.4:4000" $IMAGE
```

Note:

- *replace the IP address 1.2.3.4 in SERVICE_ENDPOINT with your actual IP*
- *wait 10 seconds until the container is started; you can use*
docker logs -f \$CONTAINER_NAME
to check if the Usage Policy of the container is already shown in the logs to confirm setup completed

2) get Bearer Token to access Semantic Container

```
APP_KEY=`docker logs $CONTAINER_NAME 2>&1 | grep APP_KEY | awk -F " " '{print $NF}'`; \  
APP_SECRET=`docker logs $CONTAINER_NAME 2>&1 | grep APP_SECRET | awk -F " " '{print $NF}'`; \  
export ADMIN_TOKEN=`curl -s -d grant_type=client_credentials -d client_id=$APP_KEY \  
-d client_secret=$APP_SECRET -d scope=admin \  
-X POST $SEMCON_URL/oauth/token | jq -r '.access_token'`
```

Note:

- *verify token was generated successfully with*
echo \$ADMIN_TOKEN

3) create a DID for the Semantic Container

```
SC_DID=`oydid sc_init -l $SEMCON_URL --token $ADMIN_TOKEN \  
--doc-pwd passphrase1 --rev-pwd passphrase2 | \  
jq -r '.did'`
```

Note:

- *display DID and DID document with the following command*
oydid --w3c-did \$SC_DID | jq

- *in the example above for simplicity a passphrase is used (--doc-pwd, --rev-pwd); it is also possible to use base58 encoded keys stored in a file (using --doc-key, --rev-key)*

4) write to Semantic Container using the DID and local private key / passphrase

```
# request token using "oydid sc_token"
TOKEN=`oydid sc_token $SC_DID --doc-pwd passphrase1 | jq -r '.access_token'`
# write data
echo '{"hello":"world"}' | \
  curl -s -H "Content-Type: application/json" -H "Authorization: Bearer $TOKEN" \
  -d @- -X POST $SEMCON_URL/api/data
```

5) create new DID for a specific record

```
REC_DID=`echo '{"service_endpoint":"/api/data/1?p=id", "scope":"read"}' | \
  oydid sc_create $SC_DID --token $ADMIN_TOKEN \
  --doc-pwd=documentpwd --rev-pwd=revocationpwd | \
  jq -r '.did'`
```

6) read from Semantic Container using the DID and local private key / passphrase

```
# request token using "oydid sc_token"
REC_TOKEN=`oydid sc_token $REC_DID --doc-pwd documentpwd | jq -r '.access_token'`
curl -s -H "Authorization: Bearer $REC_TOKEN" "$SEMCON_URL/api/data/1?p=id" | jq
# read data
curl -s -H "Authorization: Bearer $ADMIN_TOKEN" \
  "$SEMCON_URL/api/data/1?p=id&f=validation" | jq
```