Cashu Book of NUTs

TODO

Abstract

This is the abstract.
It consists of two paragraphs.

Contents

Welcome to the Cashu Book of NUTs. This book is nuts. It is a streamlined guide to the Cashu Notation, Usage, and Terminology (NUTs).

I want to be clear: I didn't write the NUTs. All the credit goes to the original authors and contributors of these notes. My contribution has been to organize these NUTs into a single larger document.

Thanks for picking up this book! I hope it helps you navigate the exciting waters of Cashu more easily and encourages you to dive deeper into this innovative project. Let's explore and build the future of decentralized communication together!

Git Commit

To provide readers with the most up-to-date information, this page showcases the latest git commit from the cashubtc/nuts repository on GitHub. This commit log offers a snapshot of the most recent changes, updates, and enhancements made to the Notation, Usage, and Terminology (NUTs). By incorporating this information, readers can gain insight into the ongoing development and evolution of Cashu ecash, ensuring they are informed about the latest contributions and modifications from the community. This inclusion underscores the dynamic nature of the project and highlights the collaborative efforts driving its progress.

```
commit 8da1abdef2fdb7f7251e824e4db7398135aa1802
Author: Rob Woodgate <robwoodgate@users.noreply.github.com>
Date:   Mon Feb 16 12:53:36 2026 +0000

    NUTXX - ECDH-derived Pay-to-Blinded-Key (P2BK) (#300)

    * NUTXX - Pay to Blinded Key (P2BK) - ECDH

    * Remove keyset_id from blinding factor calculation, rework test vectors

    * adds a line break

    * Set NUT number to 28

    * Apply suggestions from code review

    * clarify sender public key protection

    * remove redundant line - we spell our the derivation process
```

Mandatory NUTs

These NUTs `MUST` be implemented by each wallet and mint.

NUT-00: Notation, Utilization, and Terminology

`mandatory`

This document details the notation and models used throughout the specification and lays the groundwork for understanding the basic cryptography used in the Cashu protocol.

- Sending user: `Alice`
- Receiving user: `Carol`
- Mint: `Bob`

Blind Diffie-Hellmann key exchange (BDHKE)

Variables

- `G` elliptic curve generator point

Bob (mint)

- `k` private key of mint (one for each amount)
- `K` public key corresponding to `k`
- `C_` blind signature (on `B_`)

Alice (user)

- `x` UTF-8-encoded random string (secret message), corresponds to point `Y = hash_to_curve(x)` on curve
- `r` blinding factor (scalar)
- `B_` blinded message (curve point)
- `C` unblinded signature (curve point)

`hash_to_curve(x: bytes) -> curve point Y`

Deterministically maps a message to a public key point on the secp256k1 curve, utilizing a domain separator to ensure uniqueness.

`Y = PublicKey('02' || SHA256(msg_hash || counter))` where `msg_hash` is `SHA256(DOMAIN_SEPARATOR || x)`

- `Y` derived public key
- `DOMAIN_SEPARATOR` constant byte string `b"Secp256k1_HashToCurve_Cashu_"`
- `x` message to hash
- `counter` uint32 counter(byte order little endian) incremented from 0 until a point is found that lies on the curve

Protocol

- Mint `Bob` publishes public key `K = kG`
- `Alice` picks secret `x` and computes `Y = hash_to_curve(x)`
- `Alice` sends to `Bob`: `B_ = Y + rG` with `r` being a random blinding factor (**blinding**)
- `Bob` sends back to `Alice` blinded key: `C_ = kB_` (these two steps are the DH key exchange) (**signing**)
- `Alice` can calculate the unblinded key as `C_ - rK = kY + krG - krG = kY = C` (**unblinding**)
- Alice can take the pair `(x, C)` as a token and can send it to `Carol`.
- `Carol` can send `(x, C)` to `Bob` who then checks that `k*hash_to_curve(x)== C` (**verification**), and if so treats it as a valid spend of a token, adding `x` to the list of spent secrets.

Notation And Conventions

These specifications use elliptic curve public key cryptography over the field and curve parameters defined by secp256k1.

Types

Values in these specification are one of:

- Scalars: integers modulo the curve order `n`
- Curve points: elements of the elliptic curve group (including the identity point at infinity)
- Byte sequences

Byte Operations And Encodings

- Concatenation `||` appends one byte sequence to another.
- All operations such as point addition, scalar multiplication, hashing, are performed on the underlying mathematical values, not on their serialised byte encodings.
- When a curve point is serialised as `hex_str`, it is the hex encoding of the SEC1 compressed public key bytes, unless a NUT specifies a different encoding.
- Integer byte order and fixed widths are defined by each NUT where relevant, if a NUT is silent, it must not be assumed.

Hashing

- `SHA256(m)` denotes the SHA 256 hash of byte sequence `m`.
- Any hash to curve function must specify its domain separator and exact procedure, including how counters are encoded and incremented.

0.1 - Models

`BlindedMessage`

An encrypted ("blinded") secret and an amount is sent from `Alice` to `Bob` for minting tokens or for swapping tokens. A `BlindedMessage` is also called an *output*.

```
{
  "amount": int,
  "id": hex_str,
  "B_": hex_str
}
```

`amount` is the value for the requested `BlindSignature`, `id` is the requested keyset ID from which we expect a signature, and `B_` is the blinded secret message generated by `Alice`. An array `[BlindedMessage]` is also referred to as `BlindedMessages`.

`BlindSignature`

A `BlindSignature` is sent from `Bob` to `Alice` after minting tokens or after swapping tokens. A `BlindSignature` is also called a *promise*.

```
{
  "amount": int,
  "id": hex_str,
  "C_": hex_str
}
```

`amount` is the value of the blinded token, `id` is the keyset id of the mint keys that signed the token, and `C_` is the blinded signature on the secret message `B_` sent in the previous step.

`Proof`

A `Proof` is also called an *input* and is generated by `Alice` from a `BlindSignature` it received. An array `[Proof]` is called `Proofs`. `Alice` sends `Proofs` to `Bob` for melting tokens. Serialized `Proofs` can also be sent from `Alice` to `Carol`. Upon receiving the token, `Carol` deserializes it and requests a swap from `Bob` to receive new `Proofs`.

```
{
  "amount": int,
  "id": hex_str,
  "secret": str,
  "C": hex_str,
}
```

`amount` is the amount of the `Proof`, `secret` is the secret message and is a utf-8 encoded string (the use of a 64 character hex string generated from 32 random bytes is recommended to prevent fingerprinting), `C` is the unblinded signature on `secret` (hex string), `id` is the keyset id of the mint public keys that signed the token (hex string).

0.2 - Protocol

Errors

In case of an error, mints respond with the HTTP status code `400` and include the following data in their response:

```
{
  "detail": "oops",
  "code": 1337
}
```

Here, `detail` is the error message, and `code` is the error code. Error codes are to be defined in the documents concerning the use of a certain API endpoint.

0.3 - Methods

Serialization of tokens

Tokens can be serialized to send them between users `Alice` and `Carol`. Serialized tokens have a Cashu token prefix, a versioning flag, and the token. Optionally, a URI prefix for making tokens clickable on the web.

We use the following format for token serialization:

```
cashu[version][token]
```

`cashu` is the Cashu token prefix. `[version]` is a single `base64_urlsafe` character to denote the token format version.

URI tags To make Cashu tokens clickable on the web, we use the URI scheme `cashu:`. An example of a serialized token with URI tag is

```
cashu:cashuAeyJwcm9vZn...
```

V3 tokens

> *V3 tokens are deprecated and the use of the more space-efficient V4 tokens is encouraged.*

Version This token format has the `[version]` value `A`.

Format V3 tokens are base64-encoded JSON objects. The token format supports tokens from multiple mints. The JSON is serialized with a `base64_urlsafe` (base64 encoding with `/` replaced by `_` and `+` by `-`). `base64_urlsafe` strings may have padding characters (usually `=`) at the end which can be omitted. Clients need to be able to decode both cases.

```
cashuA[base64_token_json]
```

`[base64_token_json]` is the token JSON serialized in `base64_urlsafe`. `[base64_token_json]` should be cleared of any whitespace before serializing.

Token format The deserialized `base64_token_json` is

```
{
  "token": [
    {
      "mint": str,
      "proofs": Proofs
    },
    ...
  ],
  "unit": str <optional>,
  "memo": str <optional>
}
```

`mint` is the mint URL. The mint URL must be stripped of any trailing slashes (`/`). `Proofs` is an array of `Proof` objects. The next two elements are only for displaying the receiving user appropriate information: `unit` is the currency unit of the token keysets (see Keysets for supported units), and `memo` is an optional text memo from the sender.

Example Below is a TokenV3 JSON before `base64_urlsafe` serialization.

```
{
  "token": [
    {
      "mint": "https://8333.space:3338",
      "proofs": [
        {
          "amount": 2,
          "id": "009a1f293253e41e",
          "secret": "407915bc212be61a77e3e6d2aeb4c727980bda51cd06a6afc29e2861768a7837",
          "C": "02bc9097997d81afb2cc7346b5e4345a9346bd2a506eb7958598a72f0cf85163ea"
        },
        {
          "amount": 8,
          "id": "009a1f293253e41e",
          "secret": "fe15109314e61d7756b0f8ee0f23a624acaa3f4e042f61433c728c7057b931be",
          "C": "029e8e5050b890a7d6c0968db16bc1d5d5fa040ea1de284f6ec69d61299f671059"
        }
      ]
    }
  ],
  "unit": "sat",
  "memo": "Thank you."
}
```

When serialized, this becomes:

```
cashuAeyJ0b2tlbiI6W3sibWludCI6Imh0dHBzOi8vODMzMy5zcGFjZTozMzM4IiwicHJvb2ZzIjpbeyJhbW91bnQiOjIsImlkIjoiMDA5YTFmMjkzMjUzZTQxZSIsInNl
```

V4 tokens

V4 tokens are a space-efficient way of serializing tokens using the CBOR binary format. All keys are single characters and hex strings are encoded in binary. V4 tokens can only hold proofs from a single mint.

Version This token format has the `[version]` value `B`.

Format Wallets serialize tokens in a `base64_urlsafe` format (base64 encoding with `/` replaced by `_` and `+` by `-`). `base64_urlsafe` strings may have padding characters (usually `=`) at the end which can be omitted. Clients need to be able to decode both cases.

```
cashuB[base64_token_cbor]
```

Token format For readability, the structure of a TokenV4 object is shown below in its equivalent JSON form.

```
{
  "m": str, // mint URL
  "u": str, // currency unit
  "d": str <optional>, // optional memo
  "t": [
    {
      "i": bytes, // keyset ID (short or long form)
      "p": [ // proofs with this keyset ID
        {
          "a": int, // amount
          "s": str, // secret
          "c": bytes, // signature
          "d": { <optional> // optional DLEQ proof
            "e": bytes,
            "s": bytes,
            "r": bytes
          },
          "w": str <optional> // optional witness
        },
        ...
      ]
    },
    ...
  ],
}
```

`m` is the mint URL. The mint URL **MUST** be normalized by stripping any trailing slashes (`/`). `u` is the currency unit of the token keysets. Supported units are defined in Keysets. `d` is an optional, human-readable memo provided by the sender.

Within the `t` (token) array, `i` denotes the keyset ID associated with the proofs contained in `p` which are grouped by `i`. `p` is an array of `Proof` objects with the original keyset ID field `id` omitted. All proofs in the corresponding `p` array MUST belong to the same keyset ID.

Unless otherwise stated, fields of type `bytes` represent byte strings in the CBOR encoding. In the original JSON representation of `Proof` objects, these values are encoded as hexadecimal strings. Implementations MUST convert between hex strings and raw byte arrays when translating between JSON and CBOR representations.

Optional fields MAY be omitted if not present. Receivers MUST ignore unknown fields to preserve forward compatibility.

Short keyset ID To reduce the size of the `i` field and the overall Token encoding, wallets **MAY** use the short keyset ID representation (`s_id`).

The short keyset ID is defined as the first 8 bytes of the full 33-byte keyset ID:

- Byte form: `s_id = id_bytes[:8]`
- Hex form: `s_id = id_hex_str[:16]`

Wallets receiving a Token **MUST** support both short and full keyset ID representations. When a short keyset ID is encountered, the wallet **MUST** resolve it to the corresponding full keyset ID before processing the contained `Proof` objects.

If a short keyset ID resolves to more than one known full keyset ID, the identifier is considered ambiguous. In this case, the wallet **MUST** fail token parsing and return an error.

The mint is unaware of the `s_id`. All API endpoints exposed by the mint use the full keyset ID.

Example Below is a TokenV4 JSON before CBOR and `base64_urlsafe` serialization.

```
{
    "t": [
        {
            "i": h'00ffd48b8f5ecf80',
            "p": [
                {
                    "a": 1,
                    "s": "acc12435e7b8484c3cf1850149218af90f716a52bf4a5ed347e48ecc13f77388",
                    "c": h'0244538319de485d55bed3b29a642bee5879375ab9e7a620e11e48ba482421f3cf',
                },
            ],
        },
        {
            "i": h'00ad268c4d1f5826',
            "p": [
                {
                    "a": 2,
                    "s": "1323d3d4707a58ad2e23ada4e9f1f49f5a5b4ac7b708eb0d61f738f48307e8ee",
                    "c": h'023456aa110d84b4ac747aebd82c3b005aca50bf457ebd5737a4414fac3ae7d94d',
                },
                {
                    "a": 1,
                    "s": "50bcbcbb7cc6406b3fa5d57d2174f4eff8b4402b176926d3a57d3c3dcbb59d57",
                    "c": h'0273129c5719e599379a974a626363c333c56cafc0e6d01abe46d5808280789c63',
                },
            ],
        },
    ],
    "m": "http://localhost:3338",
    "u": "sat",
}
```

The `h''` values are `bytes` but displayed as hex strings here.

We serialize this JSON using CBOR which can be seen here. The resulting bytes are then serialized to a string using `base64_urlsafe` and the prefix `cashuB` is added. This leaves us with the following serialized TokenV4:

```
cashuBo2F0gqJhaUgA_9SLj17PgGFwgaNhYQFhc3hAYWNjMTI0MzVlN2I4NDg0YzNjZjE4NTAxNDkyMThhZjkwZjcxNmE1MmJmNGE1ZWQzNDdlNDhlY2MxM2Y3NzM4OGF
```

Binary Token Token can be transmitted in a binary format when applicable (for example when transmitting via NFC). For this the serialised token is prepended with a prefix and a version byte.

```
utf8("craw") || utf8(<token_version>) || <serialised_token>
```

- Binary Encoding V4: `utf8("craw")|| utf8("B")|| cbor(token_v4_object)`

NUT-01: Mint public key exchange

`mandatory`

This document outlines the exchange of the public keys of the mint `Bob` with the wallet user `Alice`. `Alice` uses the keys to unblind `Bob`'s blind signatures (see NUT-00).

Description

Wallet user `Alice` receives public keys from mint `Bob` via `GET /v1/keys`. The set of all public keys for a set of amounts is called a *keyset*.

The mint responds only with its `active` keysets. Keysets are `active` if the mint will sign outputs with it. The mint will accept tokens from inactive keysets as inputs but will not sign with them for new outputs. The `active` keysets can change over time, for example due to key rotation. A list of all keysets, active and inactive, can be requested separately (see NUT-02).

Note that a mint can support multiple keysets at the same time but will only respond with the active keysets on the endpoint `GET /v1/keys`. A wallet can ask for the keys of a specific (active or inactive) keyset via the endpoint `GET /v1/keys/{keyset_id}` (see NUT-02).

Supported Currency Units

A mint may support any currency unit(s) they can mint (NUT-04) and melt(NUT-05), either directly or indirectly, including:

- `btc`: Bitcoin (Minor Unit: 8) - though use of the `sat` unit is generally preferred
- `sat`: Bitcoin's Minor Unit (ie: `100,000,000 sat = 1 bitcoin`)
- `msat`: defined as 1/1000th of a `sat` (ie: `1,000 msat = 1 sat`)
- `auth`: reserved for Blind Authentication (see NUT-22).
- ISO 4217 currency codes (eg: `usd`, `eur`, `gbp`)
- Stablecoin currency codes (eg: `usdt`, `usdc`, `eurc`, `gyen`)

  [!IMPORTANT] For Bitcoin, ISO 4217 currencies (and stablecoins pegged to those currencies), Keyset amount values **MUST** represent an amount in the Minor Unit of that currency. Where the Minor Unit of a currency is "`0`", amounts represent whole units. For example:

  - `usd` (Minor Unit: 2) `<amount_1> = 1 cent (0.01 USD)`
  - `jpy` (Minor Unit: 0) `<amount_1> = 1 JPY`
  - `bhd` (Minor Unit: 3) `<amount_1> = 1 fils (0.001 BHD)`
  - `btc` (Minor Unit: 8) `<amount_1> = 1 sat (0.00000001 BTC)`
  - `eurc` (Pegged to EUR, so Minor Unit: 2) `<amount_1> = 1 cent (0.01 EURC)`
  - `gyen` (Pegged to JPY, so Minor Unit: 0) `<amount_1> = 1 GYEN`

Keyset generation

Keysets are generated by the mint. The mint is free to use any key generation method they like. Each keyset is identified by its keyset `id` which can be computed by anyone from its public keys (see NUT-02).

Keys in Keysets are maps of the form `{<amount_1> : <mint_pubkey_1>, <amount_2> : <mint_pubkey_2>, ...}` for each `<amount_i>` of the amounts the mint `Bob` supports and the corresponding public key `<mint_pubkey_1>`, that is `K_i` (see NUT-00). The mint **MUST** use the compressed Secp256k1 public key format to represent its public keys.

Example

Request of `Alice`:

```
GET https://mint.host:3338/v1/keys
```

With curl:

```
curl -X GET https://mint.host:3338/v1/keys
```

Response `GetKeysResponse` of `Bob`:

```
{
  "keysets": [
    {
      "id": <keyset_id_hex_str>,
      "unit": <currency_unit_str>,
      "active": <bool>,
      "input_fee_ppk": <int|null>,
      "final_expiry": <unix_timestamp_int|null>
      "keys": {
        <amount_int>: <public_key_str>,
        ...
      }
    }
  ]
}
```

Here, `active` indicates whether the mint will sign new outputs with this keyset, and `input_fee_ppk` is the fee in parts per thousand (ppk) per input spent from this keyset (see NUT-02).

Example response

```
{
  "keysets": [
    {
      "id": "009a1f293253e41e",
      "unit": "sat",
      "active": true,
      "input_fee_ppk": 100,
      "final_expiry": 1896187313,
      "keys": {
          "1": "02194603ffa36356f4a56b7df9371fc3192472351453ec7398b8da8117e7c3e104",
          "2": "03b0f36d6d47ce14df8a7be9137712c42bcdd960b19dd02f1d4a9703b1f31d7513",
          "4": "0366be6e026e42852498efb82014ca91e89da2e7a5bd3761bdad699fa2aec9fe09",
          "8": "0253de5237f189606f29d8a690ea719f74d65f617bb1cb6fbea34f2bc4f930016d",
          ...
      }
    }
  ]
}
```

Note that for a keyset in an ISO 4217 currency, the key amounts represent values in the Minor Unit of that currency (keys truncated and comments added for clarity):

```
{
  "keysets": [
    {
      "id": "00a2f293253e41f9",
      "unit": "usd",
      "active": true,
      "input_fee_ppk": 100,
      "final_expiry": 1896187313,
      "keys": {
          "1": "0229...e101", // 1 cent (0.01 USD)
          "2": "03c0...7512", // 2 cents (0.02 USD)
```

```
        "4": "0376...fe00", // 4 cents (0.04 USD)
        "8": "0263...016e", // 8 cents (0.08 USD)
        ...
      }
    }
  ]
}
```

NUT-02: Keysets and fees

`mandatory`

A keyset is a set of public keys that the mint `Bob` generates and shares with its users. It refers to the set of public keys that each correspond to the amount values that the mint supports (e.g. `1, 2, 4, 8, ...`) respectively.

Each keyset indicates its keyset `id`, the currency `unit`, whether the keyset is `active`, and an `input_fee_ppk` that determines the fees for spending ecash from this keyset.

A mint can have multiple keysets at the same time. For example, it could have one keyset for each currency `unit` that it supports. Wallets should support multiple keysets. They must respect the `active` and the `input_fee_ppk` properties of the keysets they use.

Keyset properties

Keyset ID

The keyset `id` is the identifier for a specific keyset and can be derived from the public keys and the metadata of a keyset. Wallets **SHOULD** compute the keyset `id` for a given keyset themselves to verify that the mint is using the correct keyset in its responses.

The keyset `id` is part of each `Proof` object which allows wallets to identify which mint and keyset it was generated from. The keyset field `id` is also contained in the `BlindedMessages` sent to the mint and in the `BlindSignatures` returned from it. It is also included in a serialized Cashu `Token` (see NUT-00).

Active keysets

Mints can have multiple keysets at the same time but **MUST** have at least one `active` keyset (see NUT-01). The `active` property determines whether the mint allows generating new ecash from this keyset. `Proofs` from inactive keysets with `active=false` are still accepted as inputs but new outputs (`BlindedMessages` and `BlindSignatures`) **MUST** be from `active` keysets only.

To rotate keysets, a mint can generate a new active keyset and inactivate an old one. If the `active` flag of an old keyset is set to `false`, no new ecash from this keyset can be generated and the outstanding ecash supply of that keyset can be taken out of circulation as wallets rotate their ecash to active keysets.

Wallets **SHOULD** prioritize swaps with `Proofs` from inactive keysets (see NUT-03) so they can quickly get rid of them. Wallets **CAN** swap their entire balance from an inactive keyset to an active one as soon as they detect that the keyset was inactivated. When constructing outputs for a transaction, wallets **MUST** choose only `active` keysets (see NUT-00).

Fees

Keysets indicate the fee `input_fee_ppk` that is charged when a `Proof` of that keyset is spent as an input to a transaction. The fee is given in parts per thousand (ppk) per input measured in the `unit` of the keyset. The total fee for a transaction is the sum of all fees per input rounded up to the next larger integer (that that can be represented with the keyset).

As an example, we construct a transaction spending 3 inputs (`Proofs`) from a keyset with unit `sat` and `input_fee_ppk` of `100`. A fee of `100 ppk` means `0.1 sat` per input. The sum of the individual fees are 300 ppk for this transaction. Rounded up to the next smallest denomination, the mint charges `1 sat` in total fees, i.e. `fees = ceil(0.3)== 1`. In this case, the fees for spending 1-10 inputs is 1 sat, 11-20 inputs is 2 sat and so on.

Wallet transaction construction When constructing a transaction with ecash `inputs` (example: `/v1/swap` or `/v1/melt`), wallets **MUST** add fees to the inputs or, vice versa, subtract from the outputs. The mint checks the following equation:

```
sum(inputs) - fees == sum(outputs)
```

Here, `sum(inputs)` and `sum(outputs)` mean the sum of the amounts of the inputs and outputs respectively. `fees` is calculated from the sum of each input's fee and rounded up to the next larger integer:

```
def fees(inputs: List[Proof]) -> int:
  sum_fees = 0
  for proof in inputs:
    sum_fees += keysets[proof.id].input_fee_ppk
  return (sum_fees + 999) // 1000
```

Here, the `//` operator in `(sum_fees + 999)// 1000` denotes an integer division operator (aka floor division operator) that rounds down `sum_fees + 999` to the next lower integer. Alternatively, we could round up the sum using a floating point division with `ceil(sum_fees / 1000)` although it is not recommended to do so due to the non-deterministic behavior of floating point division.

Notice that since transactions can spend inputs from different keysets, the sum considers the fee for each `Proof` indexed by the keyset ID individually.

Deriving the keyset ID

Keyset ID V2 Keyset IDs are 33 byte hex strings with a version byte (two hexadecimal characters). The currently used version byte is `01`.

Keyset IDs are derived from public data. To derive the keyset ID of a keyset, execute the following steps:

```
1 - sort public keys by their amount in ascending numerical order
2 - concatenate each amount and its corresponding lowercase public key hex string (as "amount:publickey_hex") to a
    single byte array, separating each pair with a comma (",")
3 - add the lowercase UTF8-encoded unit string prefixed with "|unit:" to the byte array (e.g. "|unit:sat")
4 - If input_fee_ppk is specified and non-zero, add the UTF8-encoded string prefixed with "|input_fee_ppk:" (e.g.
    "|input_fee_ppk:100"). If input_fee_ppk is omitted, null, or 0, it MUST be omitted from the preimage.
5 - If a final expiration is specified, add the UTF8-encoded string prefixed with "|final_expiry:" (e.g.
    "|final_expiry:1896187313")
6 - HASH_SHA256 the concatenated byte array
7 - prefix it with a keyset ID version byte "01"
```

An example implementation in Python:

```
def derive_keyset_id_v2(keys: Dict[int, PublicKey], unit: str, input_fee_ppk: Optional[int], final_expiry:
    Optional[int]) -> str:
    sorted_keys = dict(sorted(keys.items()))
    keyset_id_bytes = b",".join(
        [f"{k}:{v.serialize()}".encode("utf-8") for k, v in sorted_keys.items()]
    )
    keyset_id_bytes += f"|unit:{unit}".encode("utf-8")
    if input_fee_ppk is not None and input_fee_ppk != 0:
        keyset_id_bytes += f"|input_fee_ppk:{input_fee_ppk}".encode("utf-8")
    if final_expiry is not None and final_expiry != 0:
        keyset_id_bytes += f"|final_expiry:{final_expiry}".encode("utf-8")
    return "01" + hashlib.sha256(keyset_id_bytes).hexdigest()
```

V1 Keysets (deprecated) V1 keysets are 8 bytes long, including a version byte prefix `00`.

```
1 - sort public keys by their amount in ascending order
2 - concatenate all public keys to one byte array
3 - HASH_SHA256 the concatenated public keys
4 - take the first 14 characters of the hex-encoded hash
5 - prefix it with a keyset ID version byte
```

An example implementation in Python:

```
def derive_keyset_id_v1(keys: Dict[int, PublicKey]) -> str:
    sorted_keys = dict(sorted(keys.items()))
    pubkeys_concat = b"".join([p.serialize() for p in sorted_keys.values()])
    return "00" + hashlib.sha256(pubkeys_concat).hexdigest()[:14]
```

> [!CRITICAL] Wallet implementations should reject any attempt at importing new keysets which IDs collide with any of the previously added keysets.

Keyset Final Expiry

A unix epoch number for a future point in time that represents the final expiry of the keyset. After the keyset's final expiry, the Mint is no longer obliged to fulfill promises signed with the keys from that keyset.

This effectively implies that the Mint can irrevocably remove all of the nullifiers (`Y` values/spent ecash) associated with the expired keyset.

The final expiry is optional and **MAY** be omitted or `null` if the keyset has no final-expiry.

Example: Get mint keysets

A wallet can ask the mint for a list of all keysets via the `GET /v1/keysets` endpoint.

Request of `Alice`:

```
GET https://mint.host:3338/v1/keysets
```

With curl:

```
curl -X GET https://mint.host:3338/v1/keysets
```

Response `GetKeysetsResponse` of `Bob`:

```
{
  "keysets": [
    {
      "id": <hex_str>,
      "unit": <str>,
      "active": <bool>,
      "input_fee_ppk": <int|null>,
      "final_expiry": <int|null>
    },
    ...
  ]
}
```

Here, `id` is the keyset ID, `unit` is the unit string (e.g. "sat") of the keyset, `active` indicates whether new ecash can be minted with this keyset, and `input_fee_ppk` is the fee (per thousand units) to spend one input spent from this keyset. If `input_fee_ppk` is not given, we assume it to be `0`.

Example response

```
{
  "keysets": [
    {
      "id": "015ba18a8adcd02e715a58358eb618da4a4b3791151a4bee5e968bb88406ccf76a",
      "unit": "sat",
      "active": true,
      "input_fee_ppk": 100,
      "final_expiry": 2059210353
    },
```

```
    {
      "id": "012fbb01a4e200c76df911eeba3b8fe1831202914b24664f4bccbd25852a6708f8",
      "unit": "sat",
      "active": false,
      "input_fee_ppk": 0,
      "final_expiry": null
    }
  ]
}
```

Requesting public keys for a specific keyset

To receive the public keys of a specific keyset, a wallet can call the `GET /v1/keys/{keyset_id}` endpoint where `keyset_id` is the keyset ID.

Example

Request of `Alice`:

We request the keys for the keyset `015ba18a8adcd02e715a58358eb618da4a4b3791151a4bee5e968bb88406ccf76a`.

```
GET https://mint.host:3338/v1/keys/015ba18a8adcd02e715a58358eb618da4a4b3791151a4bee5e968bb88406ccf76a
```

With curl:

```
curl -X GET https://mint.host:3338/v1/keys/015ba18a8adcd02e715a58358eb618da4a4b3791151a4bee5e968bb88406ccf76a
```

Response of `Bob` (same as NUT-01):

```
{
  "keysets": [{
    "id": "009a1f293253e41e",
    "unit": "sat",
    "active": true,
    "input_fee_ppk": 100,
    "keys": {
        "1": "02194603ffa36356f4a56b7df9371fc3192472351453ec7398b8da8117e7c3e104",
        "2": "03b0f36d6d47ce14df8a7be9137712c42bcdd960b19dd02f1d4a9703b1f31d7513",
        "4": "0366be6e026e42852498efb82014ca91e89da2e7a5bd3761bdad699fa2aec9fe09",
        "8": "0253de5237f189606f29d8a690ea719f74d65f617bb1cb6fbea34f2bc4f930016d",
        ...
    },
  }, ...
  ]
}
```

Wallet implementation notes

Wallets can request the list of keyset IDs from the mint upon startup and load only tokens from its database that have a keyset ID supported by the mint it interacts with. This also helps wallets to determine whether the mint has added a new current keyset or whether it has changed the `active` flag of an existing one.

A useful flow is:

- If we don't have any keys from this mint yet, get all keys: `GET /v1/keys` and store them
- Get all keysets with `GET /v1/keysets`
- For all new keyset returned here which we don't have yet, get it using `GET /v1/keys/{keyset_id}` and store it
- If any of the keysets has changed its `active` flag, update it in the db and use the keyset accordingly

NUT-03: Swap tokens

`mandatory`

The swap operation is the most important component of the Cashu system. A swap operation consists of multiple inputs (`Proofs`) and outputs (`BlindedMessages`). Mints verify and invalidate the inputs and issue new promises (`BlindSignatures`). These are then used by the wallet to generate new `Proofs` (see NUT-00).

The swap operation can serve multiple use cases. The first use case is that `Alice` can use it to split her tokens to a target amount she needs to send to `Carol`, if she does not have the necessary amounts to compose the target amount in her wallet already. The second one is that `Carols`'s wallet can use it to receive tokens from `Alice` by sending them as inputs to the mint and receive new outputs in return.

Swap to send

To make this more clear, we present an example of a typical case of sending tokens from `Alice` to `Carol`.

`Alice` has 64 sat in her wallet, composed of three `Proofs`, one worth 32 sat and another two worth 16 sat. She wants to send `Carol` 40 sat but does not have the necessary `Proofs` to compose the target amount of 40 sat. For that, `Alice` requests a swap from the mint and uses `Proofs` worth `[16, 16, 32]` as inputs and asks for new outputs worth `[8, 32, 8, 16]` totalling 64 sat. Notice that the first two tokens can now be combined to 40 sat. The `Proofs` that `Alice` sent `Bob` as inputs of the swap operation are now invalidated.

Note: In order to preserve privacy around the amount that a client might want to send to another user and keep the rest as change, the client **SHOULD** ensure that the list requested outputs is ordered by amount in ascending order. As an example of what to avoid, a request for outputs expressed like so: `[16, 8, 2, 64, 8]` might imply the client is preparing a payment for 26 sat; the client should instead order the list like so: `[2, 8, 8, 16, 64]` to mitigate this privacy leak to the mint.

Swap to receive

Another useful case for the swap operation follows up the example above where `Alice` has swapped her `Proofs` ready to be sent to `Carol`. `Carol` can receive these `Proofs` using the same operation by using them as inputs to invalidate them and request new outputs from `Bob`. Only if `Carol` has redeemed new outputs, `Alice` can't double-spend the `Proofs` anymore and the transaction is settled. To continue our example, `Carol` requests a swap with input `Proofs` worth `[32, 8]` to receive new outputs (of an arbitrary distribution) with the same total amount.

Example

Request of `Alice`:

```
POST https://mint.host:3338/v1/swap
```

With the data being of the form `PostSwapRequest`:

```
{
  "inputs": <Array[Proof]>,
  "outputs": <Array[BlindedMessage]>,
}
```

With curl:

```
curl -X POST https://mint.host:3338/v1/swap -d \
{
  "inputs":
    [
      {
        "amount": 2,
        "id": "009a1f293253e41e",
```

```
        "secret": "407915bc212be61a77e3e6d2aeb4c727980bda51cd06a6afc29e2861768a7837",
        "C": "02bc9097997d81afb2cc7346b5e4345a9346bd2a506eb7958598a72f0cf85163ea"
      },
      {
      ...
      }
    ],
  "outputs":
    [
      {
        "amount": 2,
        "id": "009a1f293253e41e",
        "B_": "02634a2c2b34bec9e8a4aba4361f6bf202d7fa2365379b0840afe249a7a9d71239"
      },
      {
      ...
      }
    ],
}
```

If successful, `Bob` will respond with a `PostSwapResponse`

```
{
  "signatures": <Array[BlindSignature]>
}
```

NUT-04: Mint tokens

`mandatory`

`used in: NUT-20, NUT-23`

Minting tokens is a two-step process: requesting a mint quote and minting new tokens. This document describes the general flow that applies to all payment methods, with specifics for each supported payment method provided in dedicated NUTs.

Supported methods

Method-specific NUTs describe how to handle different payment methods. The currently specified models are:

- NUT-23 for bolt11 Lightning invoices
- NUT-25 for bolt12 Lightning offers

General Flow

The minting process follows these steps for all payment methods:

1. The wallet requests a mint quote for the `unit` to mint, specifying the payment `method`.
2. The mint responds with a quote that includes a `quote` id and a payment `request`.
3. The user pays the `request` using the specified payment method.
4. The wallet then requests minting of new tokens with the mint, including the `quote` id and new `outputs`.
5. The mint verifies payment and returns blind signatures.

Common Request and Response Formats

Requesting a Mint Quote

To request a mint quote, the wallet of `Alice` makes a `POST /v1/mint/quote/{method}` request where `method` is the payment method requested (e.g., `bolt11`, `bolt12`, etc.).

```
POST https://mint.host:3338/v1/mint/quote/{method}
```

Depending on the payment method, the request structure may vary, but all methods will include at minimum:

```
{
  "unit": <str_enum[UNIT]>
  // Additional method-specific fields may be required
}
```

The mint `Bob` responds with a quote that includes some common fields for all methods:

```
{
  "quote": <str>,
  "request": <str>,
  "unit":  <str_enum[UNIT]>,
  // Additional method-specific fields will be included
}
```

Where `quote` is the quote ID, `request` is the payment request for the quote, and `unit` corresponds to the value provided in the request.

> [!CAUTION]
>
> `quote` is a **unique and random** id generated by the mint to internally look up the payment state. `quote` **MUST** remain a secret between user and mint and **MUST NOT** be derivable from the payment request. A third party who knows the `quote` ID can front-run and steal the tokens that this operation mints. To prevent this, use NUT-20 locks to enforce public key authentication during minting.

Check Mint Quote State

To check whether a mint quote has been paid, the wallet makes a `GET /v1/mint/quote/{method}/{quote_id}`.

```
GET https://mint.host:3338/v1/mint/quote/{method}/{quote_id}
```

The mint responds with the same structure as the initial quote response.

Executing a Mint Quote

After requesting a mint quote and paying the request, the wallet proceeds with minting new tokens by calling the `POST /v1/mint/{method}` endpoint.

```
POST https://mint.host:3338/v1/mint/{method}
```

The wallet includes the following common data in its request:

```
{
  "quote": <str>,
  "outputs": <Array[BlindedMessage]>
}
```

with the `quote` being the quote ID from the previous step and `outputs` being `BlindedMessages` (see NUT-00) that the wallet requests signatures on, whose sum is `amount` as requested in the quote.

The mint then responds with:

```
{
  "signatures": <Array[BlindSignature]>
}
```

where `signatures` is an array of blind signatures on the outputs.

Adding New Payment Methods

To add a new payment method (e.g., BOLT12), implement the following:

1. Define the method-specific request and response structures following the pattern above
2. Implement the three required endpoints: quote request, quote check, and mint execution
3. Update the settings to include the new method

Settings

The settings for this NUT indicate the supported method-unit pairs for minting. They are part of the info response of the mint (NUT-06) which reads:

```
{
  "4": {
    "methods": [
      <MintMethodSetting>,
      ...
    ],
    "disabled": <bool>
  }
}
```

`MintMethodSetting` indicates supported `method` and `unit` pairs and additional settings of the mint. `disabled` indicates whether minting is disabled.

`MintMethodSetting` is of the form:

```
{
  "method": <str>,
  "unit": <str>,
  "min_amount": <int|null>,
  "max_amount": <int|null>,
  "options": <Object|null>
}
```

`min_amount` and `max_amount` indicate the minimum and maximum amount for an operation of this method-unit pair. `options` are method-specific and can be defined in method-specific NUTs.

Unblinding Signatures

Upon receiving the `BlindSignatures` from the mint, the wallet unblinds them to generate `Proofs` (using the blinding factor `r` and the mint's public key `K`, see BDHKE NUT-00). The wallet then stores these `Proofs` in its database.

NUT-05: Melting tokens

`mandatory`

`used in: NUT-08, NUT-15, NUT-23`

Melting tokens is the opposite of minting tokens (see NUT-04). Like minting tokens, melting is a two-step process: requesting a melt quote and melting tokens. This document describes the general flow that applies to all payment methods, with specifics for each supported payment method provided in dedicated method-specific NUTs.

Supported methods

Method-specific NUTs describe how to handle different payment methods. The currently specified models are:

- NUT-23 for bolt11 Lightning invoices
- NUT-25 for bolt12 Lightning offers

General Flow

The melting process follows these steps for all payment methods:

1. The wallet requests a melt quote for a `request` it wants paid by the mint, specifying the payment `method` and the `unit` the wallet would like to spend
2. The mint responds with a quote that includes a `quote` id and an `amount` demanded in the requested unit
3. The wallet sends a melting request including the `quote` id and provides `inputs` of the required amount
4. The mint executes the payment and responds with the payment `state` and any method-specific proof of payment

Synchronous vs Asynchronous Processing

Synchronous (Default): The melt request blocks until payment completion. This ensures immediate finality but may result in long request times for slow payment methods.

Asynchronous (Optional): When the `prefer_async: true` param is included and supported by the mint, the request returns immediately after validation with a `"PENDING"` state. The wallet must then monitor the payment progress through polling or websocket notifications.

Common Request and Response Formats

Requesting a Melt Quote

To request a melt quote, the wallet of `Alice` makes a `POST /v1/melt/quote/{method}` request where `method` is the payment method requested (e.g., `bolt11`, `bolt12`, etc.).

```
POST https://mint.host:3338/v1/melt/quote/{method}
```

Depending on the payment method, the request structure may vary, but all methods will include at minimum:

```
{
  "request": <str>,
  "unit": <str_enum[UNIT]>
  // Additional method-specific fields will be required
}
```

The mint `Bob` responds with a quote that includes some common fields for all methods:

```
{
  "quote": <str>,
  "amount": <int>,
  "unit": <str_enum[UNIT]>,
```

```
  "state": <str_enum[STATE]>,
  "expiry": <int>
  // Additional method-specific fields will be included
}
```

Where `quote` is the quote ID, `amount` and `unit` the amount and unit that need to be provided, and `expiry` is the Unix timestamp until which the melt quote is valid.

`state` is an enum string field with possible values `"UNPAID"`, `"PENDING"`, `"PAID"`:

- `"UNPAID"` means that the request has not been paid yet.
- `"PENDING"` means that the request is currently being paid.
- `"PAID"` means that the request has been paid successfully.

Check Melt Quote State

To check whether a melt quote has been paid, the wallet makes a `GET /v1/melt/quote/{method}/{quote_id}`.

```
GET https://mint.host:3338/v1/melt/quote/{method}/{quote_id}
```

The mint responds with the same structure as the initial quote response.

Executing a Melt Quote

To execute the melting process, the wallet calls the `POST /v1/melt/{method}` endpoint.

```
POST https://mint.host:3338/v1/melt/{method}
```

The wallet includes the following common data in its request:

```
{
  "quote": <str>,
  "inputs": <Array[Proof]>,
  "prefer_async": <bool> // optional: false if omitted
  // Additional method-specific fields may be required
}
```

where `quote` is the melt quote ID and `inputs` are the proofs with a total amount sufficient to cover the requested amount plus any fees.

When `prefer_async: true` is included in the request data:

- **If the mint supports asynchronous processing:**
  1. The mint will verify the request (including proof validation)
  2. If valid, the mint responds immediately with a `200 OK` status and the quote state set to `"PENDING"`
  3. The payment processing begins in the background
  4. The wallet can poll the quote state endpoint or subscribe to websocket notifications to monitor payment progress
- **If the mint does not support asynchronous processing:**
  - The mint ignores the `prefer_async` field and processes the request synchronously
  - The response follows the standard synchronous flow with final payment state

  [!IMPORTANT] For methods that involve external payments (like Lightning), a synchronous call will block until the payment either succeeds or fails. This can take a long time. Make sure to **use no (or a very long) timeout when making this call**!

Synchronous Response For synchronous processing, the mint responds with a structure that indicates the final payment state and includes any method-specific proof of payment when successful.

Asynchronous Response For asynchronous processing when `prefer_async: true` is included in the request data, the mint responds with:

```
HTTP/1.1 200 OK
Content-Type: application/json
```

```
{
  "quote": <str>,
  "amount": <int>,
  "unit": <str_enum[UNIT]>,
  "state": "PENDING",
  "expiry": <int>
  // Additional method-specific fields may be included
}
```

The wallet should then either:

- Poll the quote state using `GET /v1/melt/quote/{method}/{quote_id}`
- Subscribe to websocket notifications for real-time updates (if supported by the mint)

Example Asynchronous Flow

1. **Request asynchronous melt:**

```
POST https://mint.host:3338/v1/melt/bolt11
Content-Type: application/json

{
  "quote": "TRmjduhIsPxd...",
  "inputs": [...],
  "prefer_async": true
}
```

2. **Immediate response (200 OK):**

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "quote": "TRmjduhIsPxd...",
  "amount": 10,
  "unit": "sat",
  "state": "PENDING",
  "expiry": 1701704757
}
```

3. **Poll for status:**

```
GET https://mint.host:3338/v1/melt/quote/bolt11/TRmjduhIsPxd...
```

4. **Final status response:**

```
{
  "quote": "TRmjduhIsPxd...",
  "request": "lnbc100n1p3kdrv5sp5...",
  "amount": 10,
  "unit": "sat",
  "fee_reserve": 2,
  "state": "PAID",
  "expiry": 1701704757,
  "payment_preimage": "c5a1ae1f639e1f4a3872e81500fd028bece7bedc1152f740cba5c3417b748c1b"
}
```

Adding New Payment Methods

To add a new payment method (e.g., BOLT12), implement the following:

1. Define the method-specific request and response structures following the pattern above
2. Implement the three required endpoints: quote request, quote check, and melt execution
3. Update the settings to include the new method
4. Document any method-specific fields or behaviors that differ from the general flow

Settings

The mint's settings for this NUT indicate the supported method-unit pairs for melting. They are part of the info response of the mint (NUT-06):

```
{
  "5": {
    "methods": [
      <MeltMethodSetting>,
      ...
    ],
    "disabled": <bool>
  }
}
```

`MeltMethodSetting` indicates supported `method` and `unit` pairs and additional settings of the mint. `disabled` indicates whether melting is disabled.

`MeltMethodSetting` is of the form:

```
{
  "method": <str>,
  "unit": <str>,
  "min_amount": <int|null>,
  "max_amount": <int|null>,
  "options": <Object|null>
}
```

`min_amount` and `max_amount` indicate the minimum and maximum amount for an operation of this method-unit pair. `options` are method-specific and can be defined in method-specific NUTs.

NUT-06: Mint information

`mandatory`

This endpoint returns information about the mint that a wallet can show to the user and use to make decisions on how to interact with the mint.

Example

Request of `Alice`:

```
GET https://mint.host:3338/v1/info
```

With the mint's response being of the form `GetInfoResponse`:


```
{
  "name": "Bob's Cashu mint",
  "pubkey": "0283bf290884eed3a7ca2663fc0260de2e2064d6b355ea13f98dec004b7a7ead99",
  "version": "Nutshell/0.15.0",
  "description": "The short mint description",
  "description_long": "A description that can be a long piece of text.",
  "contact": [
    {
      "method": "email",
      "info": "contact@me.com"
    },
    {
      "method": "twitter",
      "info": "@me"
    },
    {
      "method": "nostr",
      "info": "npub..."
    }
  ],
  "motd": "Message to display to users.",
  "icon_url": "https://mint.host/icon.jpg",
  "urls": [
    "https://mint.host",
    "http://mint8gv0sq5ul602uxt2fe0t80e3c2bi9fy0cxedp69v1vat6ruj81wv.onion"
  ],
  "time": 1725304480,
  "tos_url": "https://mint.host/tos",
  "nuts": {
    "4": {
      "methods": [
        {
          "method": "bolt11",
          "unit": "sat",
          "min_amount": 0,
          "max_amount": 10000
        }
      ],
      "disabled": false
    },
    "5": {
      "methods": [
```

```
        {
          "method": "bolt11",
          "unit": "sat",
          "min_amount": 100,
          "max_amount": 10000
        }
      ],
      "disabled": false
    },
    "7": {
      "supported": true
    },
    "8": {
      "supported": true
    },
    "9": {
      "supported": true
    },
    "10": {
      "supported": true
    },
    "12": {
      "supported": true
    }
  }
}
```

- (optional) `name` is the name of the mint and should be recognizable.
- (optional) `pubkey` is the hex pubkey of the mint.
- (optional) `version` is the implementation name and the version of the software running on this mint separated with a slash "/".
- (optional) `description` is a short description of the mint that can be shown in the wallet next to the mint's name.
- (optional) `description_long` is a long description that can be shown in an additional field.
- (optional) `contact` is an array of contact objects to reach the mint operator. A contact object consists of two fields. The `method` field denotes the contact method (like "email"), the `info` field denotes the identifier (like "contact@me.com").
- (optional) `motd` is the message of the day that the wallet must display to the user. It should only be used to display important announcements to users, such as scheduled maintenances.
- (optional) `icon_url` is the URL pointing to an image to be used as an icon for the mint. Recommended to be squared in shape.
- (optional) `urls` is the list of endpoint URLs where the mint is reachable from.
- (optional) `time` is the current time set on the server. The value is passed as a Unix timestamp integer.
- (optional) `tos_url` is the URL pointing to the Terms of Service of the mint.
- (optional) `nuts` indicates each NUT specification that the mint supports and its settings. The settings are defined in each NUT separately.

With curl:

```
curl -X GET https://mint.host:3338/v1/info
```

Optional NUTs

These NUTs can be optionally implemented by wallets and mints to further improve Cashu.

NUT-07: Token state check

`optional`

`used in: NUT-17, NUT-11, NUT-14`

With the token state check, wallets can ask the mint whether a specific proof is already spent and whether it is in-flight in a transaction. Wallets can also request the witness data that was used to spend a proof.

Token states

A proof can be in one of the following states

- A proof is `UNSPENT` if it has not been spent yet
- A proof is `PENDING` if it is being processed in a transaction (in an ongoing payment). A `PENDING` proof cannot be used in another transaction until it is `live` again.
- A proof is `SPENT` if it has been redeemed and its secret is in the list of spent secrets of the mint.

Note: Before deleting spent proofs from their database, wallets can check if the proof is `SPENT` to make sure that they don't accidentally delete an unspent proof. Beware that this behavior can make it easier for the mint to correlate the sender to the receiver.

Important: Mints **MUST** remember which proofs are currently `PENDING` to avoid reuse of the same token in multiple concurrent transactions. This can be achieved with for example mutex lock whose key is the `Proof`'s `Y`.

Use cases

Example 1: Ecash transaction When `Alice` prepares a token to be sent to `Carol`, she can mark these tokens in her database as *pending*. She can then, periodically or upon user input, check with the mint if the token is `UNSPENT` or whether it has been redeemed by `Carol` already, i.e., is `SPENT`. If the proof is not spendable anymore (and, thus, has been redeemed by `Carol`), she can safely delete the proof from her database.

Example 2: Lightning payments If `Alice`'s melt operation takes a long time to complete (for example if she requests a very slow Lightning payment) and she closes her wallet in the meantime, the next time she comes online, she can check all proofs marked as *pending* in her database to determine whether the payment is still in flight (mint returns `PENDING`), it has succeeded (mint returns `SPENT`), or it has failed (mint returns `UNSPENT`).

Example

Request of `Alice`:

```
POST https://mint.host:3338/v1/checkstate
```

With the data being of the form `PostCheckStateRequest`:

```
{
  "Ys": <Array[hex_str]>,
}
```

Where the elements of the array in `Ys` are the hexadecimal representation of the compressed point `Y = hash_to_curve(secret)` of the `Proof` to check (see NUT-00).

Response of `Bob`:

`Bob` responds with a `PostCheckStateResponse`:

```
{
  "states": [
    {
      "Y": <hex_str>,
      "state": <str_enum[STATE]>,
```

```
      "witness": <str|null>,
    },
    ...
  ]
}
```

The elements of the `states` array MUST be returned in the same order as the corresponding `Ys` checked in the request.

- `Y` corresponds to the `Proof` checked in the request.
- `state` is an enum string field with possible values `"UNSPENT"`, `"PENDING"`, `"SPENT"`
- `witness` is the serialized witness data that was used to spend the `Proof` if the token has a NUT-10 spending condition that requires a witness such as in the case of P2PK (NUT-11) or HTLCs (NUT-14).

With curl:

Request of `Alice`:

```
curl -X POST https://mint.host:3338/v1/checkstate -H 'Content-Type: application/json' -d '{
  "Ys": [
    "02599b9ea0a1ad4143706c2a5a4a568ce442dd4313e1cf1f7f0b58a317c1a355ee"
  ]
}'
```

Response of `Bob`:

```
{
  "states": [
    {
      "Y": "02599b9ea0a1ad4143706c2a5a4a568ce442dd4313e1cf1f7f0b58a317c1a355ee",
      "state": "SPENT",
      "witness": "{\"signatures\":
          [\"b2cf120a49cb1ac3cb32e1bf5ccb6425e0a8372affdc1d41912ca35c13908062f269c0caa53607d4e1ac4c8563246c4c8a869e6ee124ea826fd4
    }
  ]
}
```

Where `Y` belongs to the provided `Proof` to check in the request, `state` indicates its state, and `witness` is the witness data that was potentially provided in a previous spend operation (can be empty).

Mint info setting

The NUT-06 `MintMethodSetting` indicates support for this feature:

```
{
  "7": {
    "supported": true
  }
}
```

NUT-08: Lightning fee return

`optional`

`depends on: NUT-05`

This document describes how the overpaid Lightning fees are handled and extends NUT-05 which describes melting tokens (i.e. paying a Lightning invoice). In short, a wallet includes *blank outputs* when paying a Lightning invoice which can be assigned a value by the mint if the user has overpaid Lightning fees. This can be the case due to the unpredictability of Lightning network fees. To solve this issue, we introduce so-called *blank outputs* which are blinded messages with an undetermined value.

The problem is also described in this gist.

Description

Before requesting a Lightning payment as described in NUT-05, `Alice` produces a number of `BlindedMessage` which are similar to ordinary blinded messages but their value is yet to be determined by the mint `Bob` and are thus called *blank outputs*. The number of necessary blank outputs is `max(ceil(log2(fee_reserve)), 1)` which ensures that there is at least one output if there is any fee. If the `fee_reserve` is `0`, then the number of blank outputs is `0` as well. The blank outputs will contain the overpaid fees that will be returned by the mint to the wallet.

This code calculates the number of necessary blank outputs in Python:

```python
def calculate_number_of_blank_outputs(fee_reserve_sat: int) -> int:
    assert fee_reserve_sat >= 0, "Fee reserve can't be negative."
    if fee_reserve_sat == 0:
        return 0
    return max(math.ceil(math.log2(fee_reserve_sat)), 1)
```

Example

The wallet wants to pay an invoice with `amount := 100 000 sat` and determines by asking the mint that `fee_reserve` is `1000 sats`. The wallet then provides `101 000 sat` worth of proofs and 10 blank `outputs` to make the payment (since `ceil(log2(1000))=ceil(9.96..)=10`). The mint pays the invoice and determines that the actual fee was `100 sat`, i.e, the overpaid fee to return is `fee_return = 900 sat`. The mint splits the amount `900` into summands of `2^n` which is `4, 128, 256, 512`. The mint inserts these amounts into the blank `outputs` it received form the wallet and generates 4 new promises. The mint then returns these `BlindSignatures` to the wallet together with the successful payment status.

Wallet flow

The wallet asks the mint for the `fee_reserve` for paying a specific bolt11 invoice of value `amount` by calling `POST /v1/melt/quote` as described in NUT-05. The wallet then provides a `PostMeltBolt11Request` to `POST /v1/melt/bolt11` that has (1) proofs of the value `amount+fee+fee_reserve`, (2) the bolt11 invoice to be paid, and finally, as a new entry, (3) a field `outputs` that has `n_blank_outputs` blinded messages that are generated before the payment attempt to receive potential overpaid fees back to her.

Mint flow

Here we describe how the mint generates `BlindSignatures` for the overpaid fees. The mint `Bob` returns in `PostMeltQuoteBolt11Response` the field `change` **ONLY IF** `Alice` has previously provided `outputs` for the change **AND** if the the `inputs` provided were greater then the `total_amount_paid - fees`.

If the `overpaid_fees = input_amount - fees - total_paid` is positive, `Bob` decomposes it to amounts supported by the keyset, typically `2^n`, and imprints them into the `blank_outputs` provided by `Alice`.

`Bob` then signs these blank outputs (now with the imprinted amounts) and thus generates `BlindSignatures`. `Bob` then returns a payment status to the wallet, and, in addition, all blind signatures it generated for the overpaid fees.

Importantly, while Bob does not necessarily return the same number of blind signatures as it received blank outputs from Alice (since some of them may be of value 0), Bob **MUST** return the all blank signatures with a value greater than 0 in the same order as the blank outputs were received and should omit all blind signatures with value 0. For example, if Bob receives 10 blank outputs but the overpaid fees only occupy 4 blind signatures, Bob will only return these 4 blind signatures with the appropriate imprinted amounts and omit the remaining 6 blind signatures with value 0. Due to the well-defined order of the returned blind signatures, Alice can map the blind signatures returned from Bob to the blank outputs it provided so that she can further apply the correct unblinding operations on them.

Example

Request of Alice:

```
POST https://mint.host:3338/v1/melt/bolt11
```

With the data being of the form PostMeltBolt11Request:

```
{
  "quote": <str>,
  "inputs": <Array[Proof]>,
  "outputs": <Array[BlindedMessage]> <-- New
}
```

where the new output field carries the BlindMessages.

The mint Bob then responds with a PostMeltQuoteBolt11Response:

```
{
  "quote": <str>,
  "request": <str>,
  "amount": <int>,
  "unit": <str_enum[UNIT]>,
  "fee_reserve": <int>,
  "state": <str_enum[STATE]>,
  "expiry": <int>,
  "payment_preimage": <str|null>,
  "change": <Array[BlindSignature]> <-- New
}
```

where the new change field carries the returned BlindSignatures due to overpaid fees.

Example

Request of Alice with curl:

```
curl -X POST https://mint.host:3338/v1/melt/bolt11 -d \
'{
  "quote": "od4CN5smMMS3K3QVHkbGGNCTxfcAIyIXeq8IrfhP",
  "inputs": [
    {
      "amount": 4,
      "id": "009a1f293253e41e",
      "secret": "429700b812a58436be2629af8731a31a37fce54dbf8cbbe90b3f8553179d23f5",
      "C": "03b01869f528337e161a6768b480fcf9f75fd248b649c382f5e352489fd84fd011",
    },
    {
      "amount": 8,
      "id": "009a1f293253e41e",
      "secret": "4f3155acef6481108fcf354f6d06e504ce8b441e617d30c88924991298cdbcad",
      "C": "0278ab1c1af35487a5ea903b693e96447b2034d0fd6bac529e753097743bf73ca9",
    }
```

```
  ],
  "outputs": [
    {
      "amount": 1,
      "id": "009a1f293253e41e",
      "B_": "03327fc4fa333909b70f08759e217ce5c94e6bf1fc2382562f3c560c5580fa69f4"
    }
  ]
}'
```

Everything here is the same as in NUT-05 except for `outputs`. The `amount` field in the `BlindedMessages` here are ignored by `Bob` so they can be set to any arbitrary value by `Alice` (they should be set to a value, like `1` so potential JSON validations do not error).

If the mint has made a successful payment, it will respond the following.

Response `PostMeltQuoteBolt11Response` from `Bob`:

```
{
  "state": "PAID",
  "payment_preimage": "c5a1ae1f639e1f4a3872e81500fd028bece7bedc1152f740cba5c3417b748c1b",
  "change": [
    {
      "id": "009a1f293253e41e",
      "amount": 2,
      "C_": "03c668f551855ddc792e22ea61d32ddfa6a45b1eb659ce66e915bf5127a8657be0"
    }
  ]
}
```

The field `change` is an array of `BlindSignatures` that account for the overpaid fees. Notice that the amount has been changed by the mint. `Alice` must take these and generate `Proofs` by unblinding them as described in NUT-00 and as she does in NUT-04 when minting new tokens. After generating the `Proofs`, `Alice` stores them in her database.

Mint info setting

The NUT-06 `MintMethodSetting` indicates support for this feature:

```
{
  "8": {
    "supported": true
  }
}
```

NUT-09: Restore signatures

`optional`

`used in: NUT-13`

In this document, we describe how wallets can recover blind signatures, and with that their corresponding `Proofs`, by requesting from the mint to reissue the blind signatures. This can be used for a backup recovery of a lost wallet (see NUT-13) or for recovering the response of an interrupted swap request (see NUT-03).

Mints must store the `BlindedMessage` and the corresponding `BlindSignature` in their database every time they issue a `BlindSignature`. Wallets provide the `BlindedMessage` for which they request the `BlindSignature`. Mints only respond with a `BlindSignature`, if they have previously signed the `BlindedMessage`. Each returned `BlindSignature` also contains the `amount` and the keyset `id` (see NUT-00) which is all the necessary information for a wallet to recover a `Proof`.

Request of `Alice`:

```
POST https://mint.host:3338/v1/restore
```

With the data being of the form `PostRestoreRequest`:

```
{
  "outputs": <Array[BlindedMessages]>
}
```

Response of `Bob`:

The mint `Bob` then responds with a `PostRestoreResponse`.

```
{
  "outputs": <Array[BlindedMessages]>,
  "signatures": <Array[BlindSignature]>
}
```

The returned arrays `outputs` and `signatures` are of the same length and for every entry `outputs[i]`, there is a corresponding entry `signatures[i]`.

Mint info setting

The NUT-06 `MintMethodSetting` indicates support for this feature:

```
{
  "9": {
    "supported": true
  }
}
```

NUT-10: Spending conditions

`optional`

`used in: NUT-11, NUT-14`

An ordinary ecash token is a set of `Proofs` each with a random string `secret`. To spend such a token in a swap or a melt operation, wallets include `proofs` in their request each with a unique `secret`. To authorize a transaction, the mint requires that the `secret` has not been seen before. This is the most fundamental spending condition in Cashu, which ensures that a token can't be double-spent.

In this NUT, we define a well-known format of `secret` that can be used to express more complex spending conditions. These conditions need to be met before the mint authorizes a transaction. Note that the specific type of spending condition is not part of this document but will be explained in other documents. Here, we describe the structure of `secret` which is expressed as a JSON `Secret` with a specific format.

Spending conditions are enforced by the mint which means that, upon encountering a `Proof` where `Proof.secret` can be parsed into the well-known format, the mint can require additional conditions to be met.

Caution: If the mint does not support spending conditions or a specific `kind` of spending condition, proofs may be treated as a regular anyone-can-spend tokens. Applications need to make sure to check whether the mint supports a specific `kind` of spending condition by checking the mint's info endpoint.

Basic components

An ecash transaction, i.e., a swap or a melt operation, with a spending condition consists of the following components:

- Inputs referring to the `Proofs` being spent
- `Secret` containing the rules for unlocking a `Proof`
- Additional witness data satisfying the unlock conditions such as signatures
- Outputs referring to the `BlindMessages` with new unlock conditions to which the `Proofs` are spent to

Spending conditions are defined for each individual `Proof` and not on a transaction level that can consist of multiple `Proofs`. Similarly, spending conditions must be satisfied by providing signatures or additional witness data for each `Proof` separately. For a transaction to be valid, all `Proofs` in that transaction must be unlocked successfully.

New `Secrets` of the outputs to which the inputs are spent to are provided as `BlindMessages` which means that they are blind-signed and not visible to the mint until they are actually spent.

Well-known Secret

Spending conditions are expressed in a well-known secret format that is revealed to the mint when spending (unlocking) a token, not when the token is minted (locked). The mint parses each `Proof`'s `secret`. If it can deserialize it into the following format it executes additional spending conditions that are further specified in additional NUTs.

The well-known `Secret` stored in `Proof.secret` is a JSON of the format:

```
[
kind <str>,
  {
    "nonce": <str>,
    "data": <str>,
    "tags": [[ "key", "value1", "value2", ...],  ... ], // (optional)
  }
]
```

- `kind` is the kind of the spending condition
- `nonce` is a unique random string
- `data` expresses the spending condition specific to each kind
- `tags` hold additional data committed to and can be used for feature extensions

Tag format

The optional `tags` field, is an array of arrays of non-empty strings.

Each individual tag is an array of **ONE or more strings**. The first element of the tag array is known as the tag *name* or *key* and the subsequent string(s) are the *tag value(s)*.

Examples of valid tags:

`["pubkeys"]` - a tag with the key, `"pubkeys"` and no tag values.
`["pubkeys", "021..."]` - a tag with the key, `"pubkeys"` and ONE tag value.
`["pubkeys", "021...", "022..."]` - a tag with the key, `"pubkeys"` and TWO tag values.

Examples of invalid tags:

`[]` - a tag array must contain at least **ONE** or more strings
`["locktime", 1765300829]` - a tag must contain **strings** only
`["locktime", ""]` - a tag must contain **non-empty strings** only

Examples

Example use cases of this secret format are

- NUT-11: Pay-to-Public-Key (P2PK)
- NUT-14: Hashed Timelock Contracts (HTLCs)

Mint info setting

The NUT-06 `MintMethodSetting` indicates support for this feature:

```
{
  "10": {
    "supported": true
  }
}
```

NUT-11: Pay to Public Key (P2PK)

`optional`

`depends on: NUT-10, NUT-08`

`extended by: NUT-28`

This NUT describes Pay-to-Public-Key (P2PK) which is one kind of spending condition based on NUT-10's well-known `Secret`. Using P2PK, we can lock ecash Proofs (see NUT-00) to a receiver's ECC public key and require a Schnorr signature with the corresponding private key to unlock the ecash. The spending condition is enforced by the mint.

Caution: If the mint does not support this type of spending condition, Proofs may be treated as regular anyone-can-spend Proofs. Applications need to make sure to check whether the mint supports a specific kind of spending condition by checking the mint's NUT-06 info endpoint.

Basic Case

NUT-10 Secret `kind: P2PK`

If for a `Proof`, `Proof.secret` is a `Secret` of kind `P2PK`, the proof must be unlocked by providing a witness `Proof.witness` and one or more valid signatures in the array `Proof.witness.signatures`.

In the basic case, when spending a locked Proof, the mint requires one valid Schnorr signature in `Proof.witness.signatures` on `Proof.secret` by the public key in `Proof.secret.data`.

To give a concrete example of the basic case, to mint a locked Proof we first create a P2PK `Secret` that reads:

```
[
  "P2PK",
  {
    "nonce": "859d4935c4907062a6297cf4e663e2835d90d97ecdd510745d32f6816323a41f",
    "data": "0249098aa8b9d2fbec49ff8598feb17b592b986e62319a4fa488a3dc36387157a7",
    "tags": [["sigflag", "SIG_INPUTS"]]
  }
]
```

Here, `Secret.data` is the public key of the recipient of the locked ecash. We serialize this `Secret` to a string in `Proof.secret` and get a blind signature by the mint that is stored in `Proof.C` (see NUT-03).

The recipient who owns the private key of the public key `Secret.data` can spend this proof by providing a signature on the serialized `Proof.secret` string that is then added to `Proof.witness.signatures`:

```
{
  "amount": 1,
  "secret":
      "[\"P2PK\",{\"nonce\":\"859d4935c4907062a6297cf4e663e2835d90d97ecdd510745d32f6816323a41f\",\"data\":\"0249098aa8b9d2fbec49f
  "C": "02698c4e2b5f9534cd0687d87513c759790cf829aa5739184a3e3735471fbda904",
  "id": "009a1f293253e41e",
  "witness":
      "{\"signatures\":[\"60f3c9b766770b46caac1d27e1ae6b77c8866ebaeba0b9489fe6a15a837eaa6fcd6eaa825499c72ac342983983fd3ba3a8a41f5
}
```

Signature scheme

To spend Proofs locked with `P2PK`, the spender needs to include signatures in the Proofs used as "inputs" for the spending operation. We use `libsecp256k1`'s serialized 64 byte Schnorr signatures on the SHA256 hash of the message to sign. The message to sign is the field `Proof.secret` in the inputs, unless otherwise indicated by the `Secret.tags.sigflag` in the inputs, as detailed below.

An ecash spending operation like swap and melt can have multiple inputs and outputs. If we have more than one locked input, we either provide signatures in each input individually (for `SIG_INPUTS`) or only in the first input for the entire transaction (for `SIG_ALL`). The inputs are the `Proofs` provided in the `inputs` field and the outputs are the `BlindedMessages` in the `outputs` field in the request body (see `PostMeltRequest` in NUT-05 and `PostSwapRequest` in NUT-03).

> [!NOTE]
>
> The field `Proof.secret` contains escaped JSON for transport. The message to sign MUST be constructed using the **unescaped** secret string, eg: [“P2PK”,{“nonce”:“c7f280eb55c1e856…

Witness format

Signatures are stored in `P2PKWitness` objects and are provided in either each `Proof.witness` of all inputs separately (for `SIG_INPUTS`) or only in the first input of the transaction (for `SIG_ALL`). `P2PKWitness` is a serialized JSON string of the form

```
{
  "signatures": <Array[<hex_str>]>
}
```

The `signatures` are an array of signatures in hex and correspond to the signatures by one or more signing public keys.

Tags

More complex spending conditions can be defined in the tags in `Secret.tags`. All tags are optional. Tags are arrays with two or more strings being `["key", "value1", "value2", ...]`. We denote a specific tag in a proof by its `key`.

Supported tags are:

- `sigflag: <str_enum[SIG_FLAG]>` sets the signature flag
- `pubkeys: <hex_str>` are additional public keys (together with the one in the `data` field of the secret) that can provide signatures (*allows multiple entries*)
- `n_sigs: <int>` specifies the minimum number of Locktime Multisig public keys providing valid signatures
- `locktime: <int>` is the Unix timestamp of when the lock expires
- `refund: <hex_str>` are additional public keys that can provide signatures after `locktime` (*allows multiple entries*)
- `n_sigs_refund: <int>` specifies the minimum number of Refund Multisig public keys providing valid signatures after `locktime` expires

  > [!NOTE]
  >
  > The tag serialization type is `[<str>, <str>, ...]` but some tag values are `int`. Wallets and mints must cast types appropriately for de/serialization.

Signature flag

Signature flags are defined in the tag `Secret.tags['sigflag']`. Currently, there are two signature flags.

- `SIG_INPUTS` requires valid signatures on all inputs independently. It is the default signature flag and will be applied if the `sigflag` tag is absent.
- `SIG_ALL` requires valid signatures on all inputs and on all outputs of a transaction.

If any one input has the signature flag `SIG_ALL`, then all inputs are required to have the same kind, the flag `SIG_ALL` and the same `Secret.data` and `Secret.tags`, otherwise an error is returned.

`SIG_INPUTS` is only enforced if no input is `SIG_ALL`.

Signature flag `SIG_INPUTS`

`SIG_INPUTS` means that each `Proof` (input) requires its own signature. The signature is provided in the `Proof.witness` field of each input separately. The format of the witness was defined earlier on.

Signed inputs A `Proof` (an input) with a signature `P2PKWitness.signatures` on `secret` is the JSON (see NUT-00):

```
{
  "amount": <int>,
  "secret": <str>,
  "C": <hex_str>,
  "id": <str>,
  "witness": <P2PKWitness | str> // Signatures on "secret"
}
```

The `secret` field is **signed as a string**.

Signature flag `SIG_ALL`

`SIG_ALL` is enforced only if the following conditions are met:

- If one input has the signature flag `SIG_ALL`, all other inputs MUST have the same `Secret.data` and `Secret.tags`, and by extension, also be `SIG_ALL`.
- If one or more inputs differ from this, an error is returned.

If this condition is met, the `SIG_ALL` flag is enforced and only **the first input of a transaction requires a witness** that covers all other inputs and outputs of the transaction. All signatures by the signing public keys MUST be provided in the `Proof.witness` of the first input of the transaction.

Message aggregation for `SIG_ALL` The message to be signed depends on the type of transaction containing an input with signature flag `SIG_ALL`.

Aggregation for `swap` A swap contains `inputs` and `outputs` (see NUT-03). To provide a valid signature, the owner (or owners) of the signing public keys must concatenate the `secret` and `C` fields of all `Proofs` (inputs) with the `amount` and `B_` fields of all `BlindedMessages` (outputs, see NUT-00) to a single message string in the order they appear in the transaction. This concatenated string is then hashed and signed (see Signature scheme).

If a swap transaction has `n` inputs and `m` outputs, the message to sign becomes:

```
msg = secret_0 || C_0 || ... || secret_n || C_n || amount_0 ||  B_0 || ... || amount_m || B_m
```

Here, `||` denotes string concatenation. The `C` of each input and `B_` of each output are **hex strings** and `amount` is a UTF-encoded string.

Aggregation for `melt` For a melt transaction, the message to sign is composed of all the inputs, the quote ID being paid, and the NUT-08 blank `outputs`.

If a melt transaction has `n` inputs, `m` blank outputs, and a quote ID `quote_id`, the message to sign becomes:

```
msg = secret_0 || C_0 || ... || secret_n || C_n || amount_0 || B_0 || ... || amount_m || B_m || quote_id
```

Here, `||` denotes string concatenation. The `C` of each input and `B_` of each output are **hex strings** and `amount` is a UTF-encoded string.

Locktime Tag

The `locktime` tag signals which set of locking rules the mint should apply. There are three possible states the `locktime` tag can represent:

Permanent Lock

If the `locktime` tag is not present, or is not a valid unix time, the lock is considered "permanent".

Locktime Multisig conditions apply if the `pubkeys` tag is present, Basic Case conditions if not.

Active Lock

If the `locktime` tag is a valid unix time and the mint's local clock is less than `locktime`, the lock is "active".

Locktime Multisig conditions apply if the `pubkeys` tag is present, Basic Case conditions if not.

Expired Lock

If the `locktime` tag is a valid unix time and the mint's local clock is greater than `locktime`, the lock has "expired".

Both Locktime Multisig and Refund Multisig conditions apply if the `refund` tag is present, otherwise the proof is considered unlocked and spendable without a witness signature.

> [!NOTE] A `Proof` is considered spendable by anyone if it only requires a `secret` and a valid unblinded signature `C` to be spent (which is the default case in NUT-00).

Multisig

Cashu offers two multi-signature pathways: `Locktime MultiSig` and `Refund MultiSig`, which are activated depending on the status of the proof's `locktime` tag.

> [!NOTE] Each pathway has a self-contained set of conditions which must be satisfied for that pathway to be valid. You cannot mix/blend conditions between pathways.

Locktime MultiSig

Locktime Multisig extends the Basic Case by allowing proofs to be locked to multiple public keys. The additional locking public keys are stored in the `pubkeys` tag.

If the `pubkeys` tag is present, the `Proof` is spendable only if a valid signature is given by **at least ONE** of the public keys contained in the `Secret.data` field or the `pubkeys` tag.

If the `n_sigs` tag is a positive integer, the mint will require at least `n_sigs` of those public keys to provide a valid signature.

If the number of public keys with valid signatures is greater or equal to the number specified in `n_sigs` (or `1` if `n_sigs` is not present), the transaction is valid. The signatures are provided in an array of strings in the `P2PKWitness` object.

Expressed as an "n-of-m" scheme, `n = n_sigs` is the number of required signatures and `m = 1 (data field)+ count(pubkeys tag keys)` is the total number of public keys that *could* sign.

> [!CAUTION]
>
> Because Schnorr signatures are non-deterministic (due to auxiliary random data), we expect a minimum number of unique public keys with valid signatures instead of expecting a minimum number of signatures.

Refund MultiSig

Refund Multisig allows proofs to be *additionally spendable* by a separate set of public keys once the `locktime` has expired. These public keys are stored in the `refund` tag, and can include keys previously listed in `data` or `pubkeys`.

Locktime Multisig conditions continue to apply, and the proof can continue to be spent according to Locktime Multisig rules.

In addition, the `Proof` can be spent if a valid signature is given by **at least ONE** of the public keys contained in the `refund` tag.

If the `n_sigs_refund` tag is a positive integer, the mint will require at least `n_sigs_refund` of those `refund` public keys to provide a valid signature.

If the number of `refund` public keys with valid signatures is greater or equal to the number specified in `n_sigs_refund` (or `1` if `n_sigs_refund` is not present), the transaction is valid. The signatures are provided in an array of strings in the `P2PKWitness` object.

Expressed as an "n-of-m" scheme, `n = n_sigs_refund` is the number of required signatures and `m = count(refund tag keys)` is the total number of `refund` keys that *could* sign.

> [!CAUTION]
>
> Because Schnorr signatures are non-deterministic (due to auxiliary random data), we expect a minimum number of unique public keys with valid signatures instead of expecting a minimum number of signatures.

Complex Example

This is an example `Secret` that locks a `Proof` with a Pay-to-Pubkey (P2PK) condition that requires 2-of-3 signatures from the public keys in the `data` field and the `pubkeys` tag.

If the `locktime` has passed, the `Proof` continues to be spendable with 2-of-3 signatures from the public keys in the `data` field and the `pubkeys` tag. But now it **ALSO** becomes spendable with a single signature from any ONE of the public keys in the `refund` tag.

The signature flag `sigflag` indicates that signatures are necessary on the `inputs` and the `outputs` of the transaction this `Proof` is spent by.


```
[
  "P2PK",
  {
    "nonce": "da62796403af76c80cd6ce9153ed3746",
    "data": "033281c37677ea273eb7183b783067f5244933ef78d8c3f15b1a77cb246099c26e",
    "tags": [
      ["sigflag", "SIG_ALL"],
      ["n_sigs", "2"],
      ["locktime", "1689418329"],
      [
        "refund",
        "033281c37677ea273eb7183b783067f5244933ef78d8c3f15b1a77cb246099c26e",
        "02e2aeb97f47690e3c418592a5bcda77282d1339a3017f5558928c2441b7731d50"
      ],
      [
        "pubkeys",
        "02698c4e2b5f9534cd0687d87513c759790cf829aa5739184a3e3735471fbda904",
        "023192200a0cfd3867e48eb63b03ff599c7e46c8f4e41146b2d281173ca6c50c54"
      ]
    ]
  }
]
```


Use cases

The following use cases are unlocked using P2PK:

- Publicly post locked ecash that can only be redeemed by the intended receiver
- Final offline-receiver payments that can't be double-spent when combined with an offline signature check mechanism like DLEQ proofs
- Receiver of locked ecash can defer and batch multiple mint round trips for receiving proofs (requires DLEQ)
- Ecash that is owned by multiple people via the multisignature abilities
- Atomic swaps when used in combination with the locktime feature

Mint info setting

The NUT-06 `MintMethodSetting` indicates support for this feature:

```
{
  "11": {
    "supported": true
  }
}
```

NUT-12: Offline ecash signature validation

`optional`

In this document, we present an extension of Cashu's crypto system to allow a user `Alice` to verify the mint `Bob`'s signature using only `Bob`'s public keys. We explain how another user `Carol` who receives ecash from `Alice` can execute the DLEQ proof as well. This is achieved using a Discrete Log Equality (DLEQ) proof. Previously, `Bob`'s signature could only be checked by himself using his own private keys (NUT-00).

The DLEQ proof

The purpose of this DLEQ is to prove that the mint has used the same private key a for creating its public key `A` (NUT-01) and for signing the BlindedMessage `B'`. `Bob` returns the DLEQ proof additional to the blind signature `C'` for a mint or swap operation.

The complete DLEQ proof reads

```
# DLEQ Proof

(These steps occur when Bob returns C')

Bob:
r = random nonce
R1 = r*G
R2 = r*B'
e = hash(R1,R2,A,C')
s = r + e*a
return e, s

Alice:
R1 = s*G - e*A
R2 = s*B' - e*C'
e == hash(R1,R2,A,C')

If true, a in A = a*G must be equal to a in C' = a*B'
```

Hash function

The hash function `hash(x: <Array<[PublicKey]>)-> bytes` generates a deterministic SHA256 hash for a given input list of `PublicKey`. The uncompressed (32+32+1)-byte hexadecimal representations (130 characters) of each `PublicKey` is concatenated before taking the SHA256 hash.

```python
def hash_e(*publickeys: PublicKey) -> bytes:
    e_ = ""
    for p in publickeys:
        _p = p.serialize(compressed=False).hex()
        e_ += str(_p)
    e = hashlib.sha256(e_.encode("utf-8")).digest()
    return e
```

> [!NOTE] For examples of valid DLEQ proofs, see the test vectors.

Mint to user: DLEQ in `BlindSignature`

The mint produces these DLEQ proofs when returning `BlindSignature`'s in the responses for minting (NUT-04) and swapping (NUT-03) tokens. The `BlindSignature` object is extended in the following way to include the DLEQ proof:

```
{
  "id": <str>,
  "amount": <int>,
  "C_": <str>,
  "dleq": { <-- New: DLEQ proof
    "e": <str>,
    "s": <str>
  }
}
```

`e` and `s` are the DLEQ proof.

User to user: DLEQ in `Proof`

In order for `Alice` to communicate the DLEQ to another user `Carol`, we extend the `Proof` (see NUT-00) object and include the DLEQ proof. As explained below, we also need to include the blinding factor `r` for the proof to be convincing to another user `Carol`.

```
{
  "id": <str>,
  "amount": <int>,
  "secret": <str>,
  "C": <str>,
  "dleq": { <-- New: DLEQ proof
    "e": <str>,
    "s": <str>,
    "r": <str>
  }
}
```

`e` and `s` are the challenge and response of the DLEQ proof returned by `Bob`, `r` is the blinding factor of `Alice` that was used to generate the `Proof`. `Alice` serializes these proofs like any other in a token (see NUT-00) to send it to another user `Carol`.

> [!IMPORTANT]
>
> **Privacy:** The blinding factor `r` should not be shared with the mint or otherwise, the mint will be able to associate the `BlindSignature` with the `Proof`.

Alice (minting user) verifies DLEQ proof

When minting or swapping tokens, `Alice` receives DLEQ proofs in the `BlindSignature` response from the mint `Bob`. `Alice` checks the validity of the DLEQ proofs for each ecash token she receives via the equations:

```
R1 = s*G - e*A
R2 = s*B' - e*C'
e == hash(R1,R2,A,C') # must be True
```

Here, the variables are

- `A` – the public key `Bob` used to sign this Proof
- `(e, s)` – the DLEQ proof returned by `Bob`
- `B'` – `Alice`'s `BlindedMessage`
- `C'` – `Bob`'s `BlindSignature` on `B'`

In order to execute the proof, `Alice` needs `e, s` that are returned in the `BlindSignature` by `Bob`. `Alice` further needs `B'` (the `BlindedMessage` `Alice` created and `Bob` signed) and `C'` (the blind signature in the `BlindSignature` response) from `Bob`, and `A` (the public key of `Bob` with which he signed the BlindedMessage). All these values are available to `Alice` during or after calling the mint and swap operations.

If a DLEQ proof is included in the mint's `BlindSignature` response, wallets **MUST** verify the DLEQ proof.

Carol (another user) verifies DLEQ proof

`Carol` is a user that receives `Proofs` in a token from another user Alice. When `Alice` sends `Proofs` with DLEQ proofs to `Carol` or when `Alice` posts the `Proofs` publicly, `Carol` can validate the DLEQ proof herself and verify `Bob`'s signature without having to talk to `Bob`. `Alice` includes the following information in the `Proof` (see above):

- `(x, C)` – the ecash `Proof`
- `(e, s)` – the DLEQ proof revealed by `Alice`
- `r` – `Alice`'s blinding factor

Here, `x` is the Proof's secret, and `C` is the mint's signature on it. To execute the DLEQ proof like `Alice` did above, `Carol` needs `(B', C')` which she can compute herself using the blinding factor `r` that she receives from `Alice`.

To verify the DLEQ proof of a received token, `Carol` needs to reconstruct `B'` and `C'` using the blinding factor `r` that `Alice` has included in the `Proof` she sent to `Carol`. Since `Carol` now has all the necessary information, she can execute the same equations to verify the DLEQ proof as `Alice` did:

```
Y = hash_to_curve(x)
C' = C + r*A
B' = Y + r*G

R1 = ... (same as Alice)
```

If a DLEQ proof is included in a received token, wallets **MUST** verify the proof.

Mint info setting

The NUT-06 `MintMethodSetting` indicates support for this feature:

```
{
  "12": {
    "supported": true
  }
}
```

NUT-13: Deterministic Secrets

`optional`

`depends on: NUT-09`

In this document, we describe the process that allows wallets to recover their ecash balance with the help of the mint using a familiar 12 word seed phrase (mnemonic). This allows us to restore the wallet's previous state in case of a device loss or other loss of access to the wallet. The basic idea is that wallets that generate the ecash deterministically can regenerate the same tokens during a recovery process. For this, they ask the mint to reissue previously generated signatures using NUT-09.

Deterministic secret derivation

An ecash token, or a `Proof`, consists of a `secret` generated by the wallet, and a signature `C` generated by the wallet and the mint in collaboration. Here, we describe how wallets can deterministically generate the `secrets` and blinding factors `r` necessary to generate the signatures `C`.

The wallet generates a `seed` derived from a 12-word BIP39 `mnemonic` seed phrase that the user stores in a secure place. The wallet uses the `seed`, to derive deterministic values for the `secret` and the blinding factors `r` for every new ecash token that it generates.

In order to do this, the wallet keeps track of a `counter_k` for each `keyset_k` it uses. The index `k` indicates that the wallet **MUST** keep track of a separate counter for each keyset `k` it uses. The wallet **MUST** keep track of multiple keysets for every mint it interacts with.

Versioned Secret Derivation The secret derivation method depends on the keyset ID version the wallet derives secrets and blinding factors for.

- **Keyset V2** (keyset IDs starting with `01`): Use HMAC-SHA256 Derivation
- **Keyset V1 (deprecated)** (keyset IDs starting with `00`): Use BIP32 Legacy Derivation

HMAC-SHA256 Derivation

For keysets with version byte `01`, the wallet uses `counter_k`, `seed` and `keyset_id` as inputs to a Key Derivation Function (KDF) which output is used to derive `secret` and `r`.

The HMAC-SHA256 KDF is built as the following:

1. `message = b"Cashu_KDF_HMAC_SHA256" || keyset_id_bytes || counter_k_bytes || derivation_type_byte`, where:
   - `"Cashu_KDF_HMAC_SHA256"` is the domain separation or purpose string, encoded to bytes as UTF-8.
   - `keyset_id_bytes` are the raw bytes of `keyset_id` (hex decoded).
   - `counter_k_bytes` is the counter encoded as an unsigned 64-bit integer in big-endian format.
   - `derivation_type_byte` is exactly 1 byte specifying the type of derivation required:
     - `0x00` for secrets
     - `0x01` for blinded messages
2. `hmac_digest = HMAC_SHA256(seed, message)`, where `HMAC_SHA256` is the hash-based message authentication code using SHA-256 as the hashing algorithm.
3. `secret = hmac_digest` and `blinding_factor = hmac_digest % N`.

Code Examples

Versioned Secret Derivation Below are code examples with keyset version-dependent derivation.

Python:

```
import hmac
import hashlib
```

```python
def derive_secret_and_r(seed: bytes, keyset_id: str, counter_k: int):
    """
    Derive secret and blinding factor using appropriate method based on keyset version
    """
    # Determine keyset version from first two characters
    keyset_version = keyset_id[:2]

    if keyset_version == "00":
        # Use legacy BIP32 derivation for version 00
        return derive_secret_and_r_bip32(seed, keyset_id, counter_k)
    elif keyset_version == "01":
        # Use HMAC-SHA256 derivation for version 01
        return derive_secret_and_r_hmac(seed, keyset_id, counter_k)
    else:
        raise ValueError(f"Unsupported keyset version: {keyset_version}")

def derive_secret_and_r_hmac(seed: bytes, keyset_id: str, counter_k: int):
    """
    HMAC-SHA256 derivation for keyset version 01

    Semantics:
      - secret = HMAC-SHA256(seed, msg || 0x00)
      - r      = OS2IP(HMAC-SHA256(seed, msg || 0x01)) mod N
      - reject r == 0 (astronomically unlikely)
    """
    # secp256k1 scalar field (group) order
    SECP256K1_N = int(
        "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141", 16
    )

    # Step 1: Create the message
    message = b"Cashu_KDF_HMAC_SHA256" + bytes.fromhex(keyset_id) + counter_k.to_bytes(8, 'big')

    # Step 2: Compute HMAC-SHA256
    secret = hmac.new(seed, message + b"\x00", hashlib.sha256).digest()
    blinding_factor_digest = hmac.new(seed, message + b"\x01", hashlib.sha256).digest()

    # Step 3: Interpret digest as integer and reduce mod N
    x = int.from_bytes(blinding_factor_digest, "big", signed=False)
    r = x % SECP256K1_N

    if r == 0:
        raise RuntimeError("Derived invalid blinding scalar r == 0")

    return secret, r

def derive_secret_and_r_bip32(seed: bytes, keyset_id: str, counter_k: int):
    """
    Legacy BIP32 derivation for keyset version 00
    """
    # Convert seed to mnemonic and derive master key
    bip32 = BIP32.from_seed(seed)

    # Calculate keyset_id_int for BIP32 derivation path
    keyset_id_int = int.from_bytes(bytes.fromhex(keyset_id), "big") % (2**31 - 1)

    # Derive secret and r using BIP32 paths
```

```
    secret_path = f"m/129372'/0'/{keyset_id_int}'/{counter_k}'/0"
    r_path = f"m/129372'/0'/{keyset_id_int}'/{counter_k}'/1"

    secret = bip32.get_privkey_from_path(secret_path)
    r = bip32.get_privkey_from_path(r_path)

    return secret, r
```

TypeScript:

```
import * as crypto from "crypto";
// secp256k1 scalar field (group) order
const SECP256K1_N = BigInt(
  "0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141",
);

function deriveSecretAndR(seed: Buffer, keysetId: string, counterK: number) {
  // Determine keyset version from first two characters
  const keysetVersion = keysetId.substring(0, 2);

  if (keysetVersion === "00") {
    // Use legacy BIP32 derivation for version 00
    return deriveSecretAndRBip32(seed, keysetId, counterK);
  } else if (keysetVersion === "01") {
    // Use HMAC-SHA256 derivation for version 01
    return deriveSecretAndRHmac(seed, keysetId, counterK);
  } else {
    throw new Error(`Unsupported keyset version: ${keysetVersion}`);
  }
}

function deriveSecretAndRHmac(
  seed: Buffer,
  keysetId: string,
  counterK: number,
) {
  // Step 1: Create message
  const counterBuffer = Buffer.alloc(8);
  counterBuffer.writeBigUInt64BE(BigInt(counterK));
  const message = Buffer.concat([
    Buffer.from("Cashu_KDF_HMAC_SHA256"),
    Buffer.from(keysetId, "hex"),
    counterBuffer,
  ]);

  const secretDerivation = Buffer.from([0]);
  const blindingFactorDerivation = Buffer.from([1]);

  // Step 2: Compute HMAC-SHA256
  const secret = crypto
    .createHmac("sha256", seed)
    .update(Buffer.concat([message, secretDerivation]))
    .digest();
  const blindingFactorDigest = crypto
    .createHmac("sha256", seed)
    .update(Buffer.concat([message, blindingFactorDerivation]))
    .digest();
```

```
  // Step 3: OS2IP + modulo reduction
  const x = BigInt("0x" + blindingFactorDigest.toString("hex"));
  const r = x % SECP256K1_N;

  if (r === 0n) {
    throw new Error("Derived invalid blinding scalar r == 0");
  }

  return { secret, r };
}

function deriveSecretAndRBip32(
  seed: Buffer,
  keysetId: string,
  counterK: number,
) {
  // Legacy BIP32 derivation for version 00
  // Implementation would use BIP32 library (e.g., bip32, bitcoinjs-lib)
  // Calculate keyset_id_int for BIP32 derivation path
  const keysetIdInt = BigInt(`0x${keysetId}`) % BigInt(2 ** 31 - 1);

  // This is pseudocode - actual implementation depends on BIP32 library
  const secretPath = `m/129372'/0'/${keysetIdInt}'/${counterK}'/0`;
  const rPath = `m/129372'/0'/${keysetIdInt}'/${counterK}'/1`;

  // const secret = bip32.derivePath(secretPath).privateKey;
  // const r = bip32.derivePath(rPath).privateKey;

  // Return placeholder for demonstration
  throw new Error(
    "BIP32 derivation requires additional library implementation",
  );
}
```

Note: See the test vectors.

Legacy Derivation (For Keyset Version `00`)

> [!NOTE] This derivation method is used for keysets with version `00` (legacy keysets). Wallets **MUST** use this method when working with keysets that have IDs starting with `00`.

BIP32 derivation paths are used. The derivation path depends on the keyset ID of `keyset_k`, and the `counter_k` of that keyset.

- Purpose′ = `129372'` (UTF-8 for)
- Coin type′ = Always `0'`
- Keyset id′ = Keyset ID represented as an integer (`keyset_k_int`)
- Coin counter′ = `counter'` (this value is incremented)
- `secret` or `r` = `0` or `1`

```
m / 129372' / 0' / keyset_k_int' / counter' / secret||r
```

This results in the following derivation paths:

```
secret_derivation_path = `m/129372'/0'/{keyset_k_int}'/{counter_k}'/0`
r_derivation_path = `m/129372'/0'/{keyset_id_k_int}'/{counter_k}'/1`
```

Here, `{keyset_k_int}` and `{counter_k}` are the only variables that can change. `keyset_id_k_int` is an integer representation (see below) of the keyset ID the token is generated with. This means that the derivation path is unique for each keyset. Note that the coin type is always `0'`, independent of the unit of the ecash.

[!NOTE] For examples, see the test vectors.

Counter The wallet starts with `counter_k := 0` upon encountering a new keyset and increments it by `1` every time it has successfully minted new ecash with this keyset. The wallet stores the latest `counter_k` in its database for all keysets it uses. Note that we have a `counter` (and therefore a derivation path) for each keyset `k`. We omit the keyset index `k` in the following of this document.

When encountering keysets with different versions, wallets **MUST** use the appropriate derivation method based on the keyset ID version and retain the existing `counter_k` value for each keyset to ensure consistent restore support across wallet implementations.

Keyset ID to Integer Mapping (deprecated)

[!CAUTION] This mapping is deprecated and unsafe to use due to its small keyspace.

The integer representation `keyset_id_int` of a keyset is calculated from its hexadecimal ID which has a length of 8 bytes or 16 hex characters. First, we convert the hex string to a big-endian sequence of bytes. This value is then modulo reduced by `2^31 - 1` to arrive at an integer that is a unique identifier `keyset_id_int`. Keyset IDs with version prefix `01` **MUST** be shortened to the first 8 bytes before conversion.

Example in Python:

```
keyset_id_int = int.from_bytes(bytes.fromhex(keyset_id_hex), "big") % (2**31 - 1)
```

Example in JavaScript:

```
keysetIdInt = BigInt(`0x${keysetIdHex}`) % BigInt(2 ** 31 - 1);
```

Restore from seed phrase

Using deterministic secret derivation, a user's wallet can regenerate the same `BlindedMessages` in case of loss of a previous wallet state. To also restore the corresponding `BlindSignatures` to fully recover the ecash, the wallet can either requests the mint to re-issue past `BlindSignatures` on the regenerated `BlindedMessages` (see NUT-09) or by downloading the entire database of the mint (TBD).

The wallet takes the following steps during recovery:

1. Determine the keyset version from the keyset ID
2. Generate `secret` and `r` from `counter` and `keyset` using the appropriate derivation method:
   - For keyset version `00`: Use legacy BIP32 derivation
   - For keyset version `01`: Use legacy BIP32 derivation and HMAC-SHA256 derivation (more on this below)
3. Generate `BlindedMessage` from `secret`
4. Obtain `BlindSignature` for `secret` from the mint
5. Unblind `BlindSignature` to `C` using `r`
6. Restore `Proof = (secret, C)`
7. Check if `Proof` is already spent

Generate `BlindedMessages` To generate the `BlindedMessages`, the wallet starts with a `counter := 0` and, for each increment of the `counter`, generates a `secret` and `r` using the appropriate derivation method based on the keyset version.

> [!CAUTION] We assume old wallets don't know the difference between `00` and `01` keyset ID and secret derivation: they might have generated secrets using the legacy derivation whereas the new derivation was required. Therefore, when performing a recovering for a `01` keyset, up-to-date wallets **MUST** check secrets with both derivation methods. (see Restoring batches)

For keyset version `00` (legacy):

```
secret = bip32.get_privkey_from_path(secret_derivation_path).hex()
r = self.bip32.get_privkey_from_path(r_derivation_path)
```

For keyset version `01` (legacy and HMAC-SHA256):

First:

```
secret = bip32.get_privkey_from_path(secret_derivation_path).hex()
r = self.bip32.get_privkey_from_path(r_derivation_path)
```

Then:

```
secret, r = derive_secret_and_r_hmac(seed, keyset_id, counter)
```

> [!NOTE] For examples, see the test vectors.

Using the `secret` string and the private key `r`, the wallet generates a `BlindedMessage`. The wallet then increases the `counter` by `1` and repeats the same process for a given batch size. It is recommended to use a batch size of 100.

The user's wallet can now request the corresponding `BlindSignatures` for theses `BlindedMessages` from the mint using the NUT-09 restore endpoint or by downloading the entire mint's database.

Generate `Proofs` Using the restored `BlindSignatures` and the `r` generated in the previous step, the wallet can unblind the signature to `C`. The triple `(secret, C, amount)` is a restored `Proof`.

Check `Proofs` states If the wallet used the restore endpoint NUT-09 for regenerating the `Proofs`, it additionally needs to check for the `Proofs` spent state using NUT-07. The wallet deletes all `Proofs` which are already spent and keeps the unspent ones in its database.

Restoring batches

Usually, the user won't remember the last state of `counter` when starting the recovery process. Therefore, wallets need to know how far they need to increment the `counter` during the restore process to be confident to have reached the most recent state.

The following approach is recommended:

- Set `counter = 0`
- Select key derivation function: legacy for `00` and HMAC-SHA256 for `01` keysets
- Restore `Proofs` in batches of 100, and increment `counter`
- Repeat restore until three consecutive batches are returned empty
- Reset `counter` to the value at the last successful restore + 1

Wallets restore `Proofs` in batches of 100. The wallet starts with a `counter=0` and increments it for every `Proof` it generated during one batch. When the wallet begins restoring the first `Proofs`, it is likely that the first few batches will only contain spent `Proofs`. Eventually, the wallet will reach a `counter` that will result in unspent `Proofs` which it stores in its database. The wallet then continues to restore until *three successive batches are returned empty by the mint*. This is to be confident that the restore process did not miss any `Proofs` that might have been generated with larger gaps in the `counter` by the previous wallet that we are restoring.

NUT-14: Hashed Timelock Contracts (HTLCs)

`optional`

`depends on: NUT-10, NUT-11`

`extended by: NUT-28`

This NUT describes the use of Hashed Timelock Contracts (HTLCs) which defines a spending condition based on NUT-10's well-known `Secret` format. Using HTLCs, ecash proofs can be locked to the hash of a preimage and a timelock. This enables use cases such as atomic swaps of ecash between users, and atomic coupling of an ecash spending condition to a Lightning HTLC.

`HTLC` spending conditions can be thought of as an extension of `P2PK` locks (see NUT-11) but with a hash lock in `Secret.data` and a new `Proof.witness.preimage` witness in the locked inputs to be spent. The `preimage` used to spend a locked Proof can be retrieved using NUT-07.

Caution: applications that rely on being able to retrieve the witness independently of the spender must check, via the mint's info endpoint, that NUT-07 is supported.

Caution: if the mint does not support this type of spending condition, proofs may be treated as regular anyone-can-spend proofs. Applications must ensure that the mint supports a specific kind of spending condition by checking the mint's info endpoint.

HTLC Locked Proof

NUT-10 Secret `kind: HTLC`

If for a `Proof`, `Proof.secret` is a `Secret` of kind `HTLC`, the hash of the lock is in `Proof.secret.data`. The preimage for unlocking the HTLC is in the witness `Proof.witness.preimage`. All additional tags from P2PK locks can also be used here, allowing a locktime, signature flag, and multisig (see NUT-11).

Here is a concrete example of a `Secret` of kind `HTLC`:

```
[
  "HTLC",
  {
    "nonce": "da62796403af76c80cd6ce9153ed3746",
    "data": "023192200a0cfd3867e48eb63b03ff599c7e46c8f4e41146b2d281173ca6c50c",
    "tags": [
      [
        "pubkeys",
        "02698c4e2b5f9534cd0687d87513c759790cf829aa5739184a3e3735471fbda904"
      ],
      ["locktime", "1689418329"],
      [
        "refund",
        "033281c37677ea273eb7183b783067f5244933ef78d8c3f15b1a77cb246099c26e"
      ]
    ]
  }
]
```

The hash lock in `Secret.data` and the preimage in `Proof.witness.preimage` are treated as 32-byte data, encoded as 64-character hexadecimal strings.

See NUT-11 for a description of the signature scheme, the additional use of signature flags, and how to require signatures from multiple public keys (Multisig).

Spending HTLC Proofs

A `Proof` with a `Secret` of kind `HTLC` can be spent in two ways.

Receiver Pathway (hash lock)

The receiver(s) listed in the `pubkeys` tag can spend the proof by providing **BOTH** of the following:

- The preimage to `Secret.data` in the `Proof.witness`
- Signature(s) as per the NUT-11 rules for **Locktime MultiSig**.

This pathway is **ALWAYS** available to the receivers, as possession of the preimage confirms performance of the Sender's wishes.

Sender Pathway (timelocked refund)

The sender(s) listed in the `refund` tag can spend the proof once the `locktime` lock has "expired" by providing signature(s) as per the NUT-11 rules for **Refund MultiSig**.

NOTE: As per the NUT-11 rules, if the `refund` tag is not present, the HTLC proof would become "anyone can spend" after the `locktime` lock has "expired". Likewise, if the `locktime` is not present, or is not a valid unix timestamp, the HTLC proof will be permanently locked and can only be spent using the Receiver (hash lock) pathway.

Hash lock

Aligned with Bitcoin's HTLC construction, the hash lock in `Secret.data` represents the **SHA-256 hash** of a 32-byte preimage.

When an HTLC-locked proof is created, the `Secret.data` field must contain:

```
hash_hex = bytes_to_hex(SHA256(preimage_bytes))
```

where:

- `preimage_bytes` is exactly 32 bytes of arbitrary data, commonly random and uniformly distributed
- `hash_hex` is the 32-byte SHA-256 digest of preimage_bytes, encoded as a 64-character lowercase hexadecimal string

To successfully spend a Proof via the **Receiver Pathway**, the spender must present the matching `preimage_bytes`, encoded as a 64-character lowercase hexadecimal string in the `Proof.witness.preimage`.

Mints and wallets **must verify** this equality before accepting the spend as valid:

```
SHA256(hex_to_bytes(Proof.witness.preimage)) == hex_to_bytes(Proof.secret.data)
```

Here is an example of a matching hash / preimage pair:

```
Proof.secret.data = 'ec4916dd28fc4c10d78e287ca5d9cc51ee1ae73cbfde08c6b37324cbfaac8bc5'
Proof.witness.preimage = '0000000000000000000000000000000000000000000000000000000000000001'
```

This hash-lock mechanism ensures that the Proof can only be spent once the secret preimage is revealed, allowing interoperability with external HTLC systems (such as Bitcoin or Lightning Network contracts).

Witness format

`HTLCWitness` is a serialized JSON string of the form

```
{
  "preimage": <hex_str>,
  "signatures": <Array[<hex_str>]>
}
```

The witness for a spent proof can be obtained with a `Proof` state check (see NUT-07).

Complex Example

This is an example `Secret` that locks a `Proof` with an HTLC condition that can be spent in either of the following ways:

Receiver Pathway - 2-of-3 signatures from the public keys in the `pubkeys` tag **PLUS** the preimage in `Proof.witness.preimage` of the hash in `Secret.data`.

Sender Pathway - One signature from the public keys in the `refund` tag, only available once the `locktime` has expired.

The signature flag `sigflag` indicates that signatures are necessary on the `inputs` and the `outputs` of the transaction this `Proof` is spent by.

```
[
  "HTLC",
  {
    "nonce": "da62796403af76c80cd6ce9153ed3746",
    "data": "023192200a0cfd3867e48eb63b03ff599c7e46c8f4e41146b2d281173ca6c50c",
    "tags": [
      ["sigflag", "SIG_ALL"],
      ["n_sigs", "2"],
      ["locktime", "1689418329"],
      [
        "refund",
        "033281c37677ea273eb7183b783067f5244933ef78d8c3f15b1a77cb246099c26e",
        "02e2aeb97f47690e3c418592a5bcda77282d1339a3017f5558928c2441b7731d50"
      ],
      [
        "pubkeys",
        "02698c4e2b5f9534cd0687d87513c759790cf829aa5739184a3e3735471fbda904",
        "0279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798",
        "0249098aa8b9d2fbec49ff8598feb17b592b986e62319a4fa488a3dc36387157a7"
      ]
    ]
  }
]
```

Mint info setting

The NUT-06 `MintMethodSetting` indicates support for this feature:

```
{
  "14": {
    "supported": true
  }
}
```

NUT-15: Partial multi-path payments

`optional`

`depends on: NUT-05`

In this document, we describe how wallets can instruct multiple mints to each pay a partial amount of a bolt11 Lightning invoice. The full payment is composed of partial payments (MPP) from multiple multi-path payments from different Lightning nodes. This way, wallets can pay a larger Lightning invoice combined from multiple smaller balances on different mints. Due to the atomic nature of MPP, either all payments will be successful or all of them will fail.

The Lightning backend of the mint must support paying a partial amount of an Invoice and multi-path payments (MPP, see BOLT 4). For example, the mint's Lightning node must be able to pay 50 sats of a 100 sat bolt11 invoice. The receiving Lightning node must support receiving multi-path payments as well.

Multimint payment execution

`Alice`'s wallet coordinates multiple MPPs on different mints that support the feature (see below for the indicated setting). For a given Lightning invoice of `amount_total`, `Alice` splits the total amount into several partial amounts `amount_total = amount_1 + amount_2 + ...` that each must be covered by her individual balances on the mints she wants to use for the payment. She constructs multiple `PostMeltQuoteBolt11Request`'s that each include the corresponding partial amount in the payment option (see below) that she sends to all mints she wants to use for the payment. The mints then respond with a normal `PostMeltQuoteBolt11Response` (see NUT-05). `Alice` proceeds to pay the melt requests at each mint simultaneously. When all mints have sent out the partial Lightning payment, she receives a successful response from all mints individually.

Melt quote

To request a melt quote with a partial `amount`, the wallet of `Alice` makes a `POST /v1/melt/quote/bolt11` similar to NUT-05.

```
POST https://mint.host:3338/v1/melt/quote/bolt11
```

The wallet `Alice` includes the following `PostMeltQuoteBolt11Request` data in its request which includes an additional (and optional) `options` object compared to the standard request in NUT-05:

```
{
  "request": <str>,
  "unit": <str_enum["sat"]>,
  "options": {
    "mpp": {
      "amount": <int>
    }
  }
}
```

Here, `request` is the bolt11 Lightning invoice to be paid, `unit` is the unit the wallet would like to pay with. `amount` is the partial amount for the requested payment in millisats (msat). The wallet then pays the returned melt quote the same way as in NUT-05.

Mint info setting

The settings returned in the info endpoint (NUT-06) indicate that a mint supports this NUT. The mint MUST indicate each `method` and `unit` that supports mpp. It can indicate this in an array of objects for multiple `method` and `unit` pairs.

`MultipathPaymentSetting` is of the form:

```
{
  [
    {
      "method": <str>,
      "unit": <str>
    },
    ...
  ]
}
```

Example `MultipathPaymentSetting`:

```
{
  "15": {
    "methods": [
      {
        "method": "bolt11",
        "unit": "sat"
      },
      {
        "method": "bolt11",
        "unit": "usd"
      }
    ]
  }
}
```

NUT-16: Animated QR codes

`optional`

This document outlines how tokens should be displayed as QR codes for sending them between two wallets.

Introduction

QR codes are a great way to send and receive Cashu tokens. Before a token can be shared as a QR code, it needs to be serialized (see NUT-00).

Static QR codes

If the serialized token is not too large (i.e. includes less than or equal to 2 proofs) it can usually be shared as a static QR code. This might not be the case if the secret includes long scripts or the token has a long memo or mint URL.

Animated QR codes

If a token is too large to be displayed as a single QR code, we use animated QR codes are based on the UR protocol. The sender produces an animated QR code from a serialized Cashu token. The receiver scans the animated QR code until the UR decoder is able to decode the token.

Resources

- Blockchain commons UR
- Typescript library bc-ur

NUT-17: WebSockets

`optional`

`depends on: NUT-07`

This NUT defines a websocket protocol that enables bidirectional communication between apps and mints using the JSON-RPC format.

Subscriptions

The websocket enables real-time subscriptions that wallets can use to receive notifications for a state change of a `MintQuoteResponse` (NUT-04), `MeltQuoteResponse` (NUT-05), `CheckStateResponse` (NUT-07).

A summary of the subscription flow is the following:

1. A wallet connects to the websocket endpoint and sends a `WsRequest` with the `subscribe` command.
2. The mint responds with a `WsResponse` containing an ok or an error.
3. If the subscription was accepted, the mint sends a `WsNotification` of the current state of the subscribed objects and whenever there is an update for the wallet's subscriptions.
4. To close a subscription, the wallet sends `WsRequest` with the `unsubscribe` command.

Specifications

The websocket is reachable via the mint's URL path `/v1/ws`:

```
https://mint.com/v1/ws
```

`NUT-17` uses the JSON-RPC format for all messages. There are three types of messages defined in this NUT.

Requests

All requests from the wallet to the mint are of the form of a `WsRequest`:

```
{
  "jsonrpc": "2.0",
  "method": <str_enum[WsRequestMethod]>,
  "params": <str_WsRequestParams>,
  "id": <int>
}
```

`WsRequestMethod` is a enum of strings with the supported commands `"subscribe"` and `"unsubscribe"`:

```
enum WsRequestMethod {
  sub = "subscribe",
  unsub = "unsubscribe",
}
```

`WsRequestParams` is a serialized JSON with the parameters of the corresponding command.

Command: Subscribe To subscribe to updates, the wallet sends a `"subscribe"` command with the following `params` parameters:

```
{
  "kind": <str_enum[SubscriptionKind]>,
  "subId": <string>,
  "filters": <string[]>
}
```

Here, `subId` is a unique uuid generated by the wallet and allows the client to map its requests to the mint's responses.

`SubscriptionKind` is an enum with the following possible values:

```
enum SubscriptionKind {
  bolt11_melt_quote = "bolt11_melt_quote",
  bolt11_mint_quote = "bolt11_mint_quote",

  bolt12_melt_quote = "bolt12_melt_quote",
  bolt12_mint_quote = "bolt12_mint_quote",

  proof_state = "proof_state",
}
```

The `filters` are an array of mint quote IDs (NUT-04), or melt quote IDs (NUT-05), or `Y`'s (NUT-07) of the corresponding object to receive updates from.

As an example, `filters` would be of the following form to subscribe for updates of three different mint quote IDs:

```
["20385fc7245...", "d06667cda9b...", "e14d8ca96f..."]
```

Note that `id` and `subId` are unrelated. The `subId` is the ID for each subscription, whereas `id` is part of the JSON-RPC spec and is an integer counter that must be incremented for every request sent over the websocket.

Important: If the subscription is accepted by the mint, the mint MUST first respond with the *current* state of the subscribed object and continue sending any further updates to it.

For example, if the wallet subscribes to a `Proof.Y` of a `Proof` that has not been spent yet, the mint will first respond with a `ProofState` with `state == "UNSPENT"`. If the wallet then spends this `Proof`, the mint would send a `ProofState` with `state == "PENDING"` and then one with `state == "SPENT"`. In total, the mint would send three notifications to the wallet.

Command: Unsubscribe The wallet should always unsubscribe any subscriptions that is isn't interested in anymore. The parameters for the `"unsubscribe"` command is only the subscription ID:

```
{
  "subId": <string>
}
```

Responses

A `WsResponse` is returned by the mint to both the `"subscribe"` and `"unsubscribe"` commands and indicates whether the request was successful:

```
{
  "jsonrpc": "2.0",
  "result": {
    "status": "OK",
    "subId": <str>
  },
  "id": <int>
}
```

Here, the `id` corresponds to the `id` in the request (as part of the JSON-RPC spec) and `subId` corresponds to the subscription ID.

Notifications

`WsNotification`'s are sent from the mint to the wallet and contain subscription data in the following format

```
{
  "jsonrpc": "2.0",
  "method": "subscribe",
  "params": {
    "subId": <str>,
    "payload": NotificationPayload
  }
}
```

`subId` is the subscription ID (previously generated by the wallet) this notification corresponds to. `NotificationPayload` carries the subscription data which is a `MintQuoteResponse` (NUT-04), a `MeltQuoteResponse` (NUT-05), or a `CheckStateResponse` (NUT-07), depending on what the corresponding `SubscriptionKind` was.

Errors

`WsErrors` for a given `WsRequest` are returned in the following format

```
{
  "jsonrpc": "2.0",
  "error": {
    "code": -32601,
    "message": "Human readable error message"
  },
  "id": "1"
}
```

Example: `ProofState` **subscription**

To subscribe to the `ProofState` of a `Proof`, the wallet establishes a websocket connection to `https://mint.com/v1/ws` and sends a `WsRequest` with a `filters` chosen to be the a `Proof.Y` value of the `Proof` (see NUT-00). Note that `filters` is an array meaning multiple subscriptions of the same `kind` can be made in the same request.

Wallet:

```
{
  "jsonrpc": "2.0",
  "id": 0,
  "method": "subscribe",
  "params": {
    "kind": "proof_state",
    "filters": [
      "02e208f9a78cd523444aadf854a4e91281d20f67a923d345239c37f14e137c7c3d"
    ],
    "subId": "Ua_IYvRHoCoF_wsZFlJ1m4gBDB--O0_6_n0zHg2T"
  }
}
```

The mint first responds with a `WsResponse` confirming that the subscription has been added.

Mint:

```
{
  "jsonrpc": "2.0",
  "result": {
    "status": "OK",
    "subId": "Ua_IYvRHoCoF_wsZFlJ1m4gBDB--O0_6_n0zHg2T"
  },
  "id": 0
}
```

The mint immediately sends the current `ProofState` of the subscription as a `WsNotification`.

Mint:

```
{
  "jsonrpc": "2.0",
  "method": "subscribe",
  "params": {
    "subId": "Ua_IYvRHoCoF_wsZFlJ1m4gBDB--00_6_n0zHg2T",
    "payload": {
      "Y": "02e208f9a78cd523444aadf854a4e91281d20f67a923d345239c37f14e137c7c3d",
      "state": "UNSPENT",
      "witness": null
    }
  }
}
```

While leaving the websocket connection open, the wallet then spends the ecash. The mint sends `WsNotification` updating the wallet about state changes of the `ProofState` accordingly:

Mint:

```
{"jsonrpc": "2.0", "method": "subscribe", "params": {"subId": "Ua_IYvRHoCoF_wsZFlJ1m4gBDB--00_6_n0zHg2T", "payload":
    {"Y": "02e208f9a78cd523444aadf854a4e91281d20f67a923d345239c37f14e137c7c3d", "state": "PENDING"}}}

{"jsonrpc": "2.0", "method": "subscribe", "params": {"subId": "Ua_IYvRHoCoF_wsZFlJ1m4gBDB--00_6_n0zHg2T", "payload":
    {"Y": "02e208f9a78cd523444aadf854a4e91281d20f67a923d345239c37f14e137c7c3d", "state": "SPENT"}}}
```

The wallet then unsubscribes.

Wallet:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "unsubscribe",
  "params": { "subId": "Ua_IYvRHoCoF_wsZFlJ1m4gBDB--00_6_n0zHg2T" }
}
```

Mint info setting

Mints signal websocket support via NUT-06 using the following setting:

```
"nuts": {
    "17": {
      "supported": [
        {
          "method": <str>,
          "unit": <str>,
          "commands": <str[]>
        },
        ...
      ]
    }
}
```

Here, `commands` is an array of the commands that the mint supports. A mint that supports all commands would return `["bolt11_mint_quote", "bolt11_melt_quote", "bolt12_mint_quote", "bolt12_melt_quote", "proof_state"]`. Supported commands are given for each method-unit pair.

Example:

```
"nuts": {
    "17": {
      "supported": [
        {
          "method": "bolt11",
          "unit": "sat",
          "commands": [
            "bolt11_mint_quote",
            "bolt11_melt_quote",
            "proof_state"
            ]
        },
      ]
    }
}
```

NUT-18: Payment Requests

`optional`

This NUT introduces a standardised format for payment requests, that supply a sending wallet with all information necessary to complete the transaction. This enables many use-cases where a transaction is better initiated by the receiver (e.g. point of sale).

Flow

1. Receiver creates a payment request, encodes it and displays it to the sender
2. Sender scans the request and constructs a matching token
3. Sender sends the token according to the transport specified in the payment request
4. Receiver receives the token and finalises the transaction

Payment Request

A Payment Request is defined as follows

```
{
  "i": str <optional>,
  "a": int <optional>,
  "u": str <optional>,
  "s": bool <optional>,
  "m": Array[str] <optional>,
  "d": str <optional>,
  "t": Array[Transport] <optional>,
  "nut10": NUT10Option <optional>,
}
```

Here, the fields are

- `i`: Payment id to be included in the payment payload
- `a`: The amount of the requested payment
- `u`: The unit of the requested payment (MUST be set if `a` is set)
- `s`: Whether the payment request is for single use
- `m`: A set of mints from which the payment is requested
- `d`: A human readable description that the sending wallet will display after scanning the request
- `t`: The method of `Transport` chosen to transmit the payment (can be multiple, sorted by preference)
- `nut10`: The required NUT-10 locking condition

Locking conditions

The payment request can include *optional* locking conditions the payee requires from the payer. For example, the payee might require a P2PK-locked token so that they can receive payments offline.

The `nut10` field specifies the payee's requested locking condition as a `NUT10Option` object. Its elements are derived from NUT-10's well-known secret. The `NUT10Option` is defined as follows:

```
{
  "k": str,
  "d": str,
  "t": Array[Array[str, str]] <optional>
}
```

- `k`: NUT-10 secret kind,
- `d`: NUT-10 secret data,
- `t`: optional NUT-10 payment tags

> [!IMPORTANT] The payee must validate the incoming tokens themselves in order to decide whether they can accept the payment. This includes checking the DLEQ proof and whether the token includes a long-enough timelock to satisfy the payee.

Transport

`Transport` specifies methods for sending the ecash to the receiver. A transport consists of a type and a target.

> [!IMPORTANT] The transport can be empty! If the transport is empty, we implicitly assume that the payment will be in-band. An example is X-Cashu where the payment is expected in the HTTP header of a request. We can only hope that the protocol you're using has a well-defined transport.

```
{
  "t": str,
  "a": str,
  "g": Array[Array[str, str]] <optional>
}
```

- `t`: type of Transport
- `a`: target of Transport
- `g`: optional tags for the Transport

Tags

Tags are an optional array of `[tag, value, value, ...]` tuples that can be used to specify additional features about the transport. A single tag can have multiple values.

Transport types

The supported transport types are described below.

Nostr

- type: `nostr`
- target: `<nprofile>`
- tags: `[["n", "17"]]`

The `n` tag specifies the NIPs the receiver supports. At least one tag value MUST be specified. For NIP-17 direct messages, the sender sends a `PaymentRequestPayload` as the message content.

HTTP POST

- type: `post`
- target: `<endpoint url>`

The execute the payment, the sender makes a `POST` request to the specified endpoint URL with the `PaymentRequestPayload` as the body.

Payment payload

If not specified otherwise, the payload sent to the receiver is a `PaymentRequestPayload` JSON serialized object as follows:

```
{
  "id": str <optional>,
  "memo": str <optional>,
  "mint": str,
  "unit": <str_enum>,
  "proofs": Array<Proof>
}
```

Here, `id` is the payment id (corresponding to `i` in request), `memo` is an optional memo to be sent to the receiver with the payment, `mint` is the mint URL from which the ecash is from, `unit` is the unit of the payment, and `proofs` is an array of proofs (see NUT-00, can also include DLEQ proofs).

Encoded Request

The payment request is serialized using CBOR, encoded in `base64_urlsafe`, together with a prefix `creq` and a version `A`:

`"creq" + "A" + base64_urlsafe(CBOR(PaymentRequest))`

Example

This is an example payment request expressed as JSON:


```
{
  "i": "b7a90176",
  "a": 10,
  "u": "sat",
  "m": ["https://nofees.testnut.cashu.space"],
  "t": [
    {
      "t": "nostr",
      "a":
          "nprofile1qy28wumn8ghj7un9d3shjtnyv9kh2uewd9hsz9mhwden5te0wfjkccte9curxven9eehqctrv5hszrthwden5te0dehhxtnvdakqqgydaqy7c
      "g": [["n", "17"]]
    }
  ]
}
```


This payment request serializes to (see here):

`creqApWF0gaNhdGVub3N0cmFheKlucHJvZmlsZTFxeTI4d3VtbjhnaGo3dW45ZDNzaGp0bnl2OWtoMnVld2Q5aHN6OW1od2RlbjV0ZTB3ZmprY2N0ZTljdXJ4dmVuOWVla`

NUT-19: Cached Responses

`optional`

To minimize the risk of loss of funds to due network errors during critical operations such as minting, swapping, and melting, we introduce a caching mechanism for successful responses. This allows wallet to replay requests on cached endpoints and allow it to recover the desired state after a network interruption.

Requests & Responses

Any Mint implementation should elect a data structure `D` that maps request objects to their respective responses. `D` should be fit for fast insertion, look-up and deletion (eviction) operations. This could be an in-memory database or a dedicated caching service like Redis.

Derive & Repeat

Upon receiving a `request` on a cached endpoint, the mint derives a unique key `k` for it which should depend on the method, path, and the payload of `request`.

The mint uses `k` to look up a `response = D[k]` and discriminates execution based on the following checks:

- If no cached `response` is found: `request` has no matching `response`. The mint processes `request` as per usual.
- If a cached `response` is found: `request` has a matching `response`. The mint returns the cached `response`.

Store

For each successful response on a cached endpoint (`status_code == 200`), the mint stores the response in `D` under key `k` (`D[k] = response`).

Expiry

The mint decides the `ttl` (Time To Live) of cached response, after which it can evict the entry from `D`.

Settings

Support for NUT-19 is announced as an extension to the `nuts` field of the `GetInfoResponse` described in NUT-06.

The entry is structured as follows:

```
"nuts": {
    ...,
    "19": {
        "ttl": <int|null>,
        "cached_endpoints": [
            {
                "method": "POST",
                "path": "/v1/mint/bolt11",
            },
            {
                "method": "POST",
                "path": "/v1/swap",
            },
            ...
        ]
    }
}
```

Where:

- `ttl` is the number of seconds the responses are cached for
- `cached_endpoints` is a list of the methods and paths for which caching is enabled.
- `path` and `method` describe the cached route and its method respectively.

If `ttl` is `null`, the responses are expected to be cached *indefinitely*.

NUT-20: Signature on Mint Quote

`optional`

`depends on: NUT-04`

This NUT defines signature-based authentication for mint quote redemption. When requesting a mint quote, clients provide a public key. The mint will then require a valid signature from the corresponding secret key to process the mint operation.

> [!CAUTION]
>
> NUT-04 mint quotes without a public key can be minted by anyone who knows the mint quote id without providing a signature.

Mint quote

To request a mint quote, the wallet of `Alice` makes a `POST /v1/mint/quote/{method}` request where `method` is the payment method requested. We present an example with the `method` being `bolt11` here.

```
POST https://mint.host:3338/v1/mint/quote/bolt11
```

The wallet of `Alice` includes the following `PostMintQuoteBolt11Request` data in its request:

```
{
  "amount": <int>,
  "unit": <str_enum["sat"]>,
  "description": <str>, // Optional
  "pubkey": <str> // Optional <-- New
}
```

with the requested `amount`,`unit`, and `description` according to NUT-04.

`pubkey` is the compressed secp256k1 public key (33 bytes, hex-encoded) that will be required for signature verification during the minting operation. The mint will only mint ecash after receiving a valid signature from the corresponding private key in the subsequent `PostMintRequest`.

> [!IMPORTANT]
>
> **Privacy:** To prevent the mint from being able to link multiple mint quotes, wallets **SHOULD** generate a unique public key for each mint quote request.

The mint `Bob` then responds with a `PostMintQuoteBolt11Response`:

```
{
  "quote": <str>,
  "request": <str>,
  "state": <str_enum[STATE]>,
  "expiry": <int>,
  "pubkey": <str> // Optional <-- New
}
```

The response is the same as in NUT-04 except for `pubkey` which has been provided by the wallet in the previous request.

Example

Request of `Alice` with curl:

```
curl -X POST http://localhost:3338/v1/mint/quote/bolt11 -d '{"amount": 10, "unit": "sat", "pubkey":
    "03d56ce4e446a85bbdaa547b4ec2b073d40ff802831352b8272b7dd7a4de5a7cac"}' -H "Content-Type: application/json"
```

Response of Bob:

```
{
  "quote": "9d745270-1405-46de-b5c5-e2762b4f5e00",
  "request": "lnbc100n1pj4apw9...",
  "state": "UNPAID",
  "expiry": 1701704757,
  "pubkey": "03d56ce4e446a85bbdaa547b4ec2b073d40ff802831352b8272b7dd7a4de5a7cac"
}
```

Signing the mint request

Message aggregation

To provide a signature for a mint request, the owner of the signing public keys must concatenate the quote ID `quote` in `PostMintQuoteBolt11Response` and the `B_` fields of all `BlindedMessages` in the `PostMintBolt11Request` (i.e., the outputs, see NUT-00) to a single message string in the order they appear in the `PostMintRequest`. This concatenated string is then hashed and signed (see Signature scheme).

> [!NOTE]
>
> Concatenating the quote ID and the outputs into a single message prevents maliciously replacing the outputs.

If a request has `n` outputs, the message to sign becomes:

```
msg_to_sign =  quote || B_0 || ... || B_(n-1)
```

Where `||` denotes concatenation, `quote` is the UTF-8 quote id in `PostMintQuoteBolt11Response`, and each `B_n` is a UTF-8 encoded hex string of the outputs in the `PostMintBolt11Request`.

Signature scheme

To mint a quote where a public key was provided, the wallet includes a signature on `msg_to_sign` in the `PostMintBolt11Request`. We use a BIP340 Schnorr signature on the SHA-256 hash of the message to sign as defined above.

Minting tokens

After requesting a mint quote and paying the request, the wallet proceeds with minting new tokens by calling the `POST /v1/mint/{method}` endpoint where `method` is the payment method requested (here `bolt11`).

```
POST https://mint.host:3338/v1/mint/bolt11
```

The wallet `Alice` includes the following `PostMintBolt11Request` data in its request

```
{
  "quote": <str>,
  "outputs": <Array[BlindedMessage]>,
  "signature": <str|null> <-- New
}
```

with the `quote` being the quote ID from the previous step and `outputs` being `BlindedMessages` as in NUT-04.

`signature` is the signature on the `msg_to_sign` which is the concatenated quote id and the outputs as defined above.

The mint responds with a `PostMintBolt11Response` as in NUT-04 if all validations are successful.

Example

Request of `Alice` with curl:

```
curl -X POST https://mint.host:3338/v1/mint/bolt11 -H "Content-Type: application/json" -d \
'{
  "quote": "9d745270-1405-46de-b5c5-e2762b4f5e00",
  "outputs": [
    {
      "amount": 8,
      "id": "009a1f293253e41e",
      "B_": "035015e6d7ade60ba8426cefaf1832bbd27257636e44a76b922d78e79b47cb689d"
    },
    {
      "amount": 2,
      "id": "009a1f293253e41e",
      "B_": "0288d7649652d0a83fc9c966c969fb217f15904431e61a44b14999fabc1b5d9ac6"
    }
  ],
  "signature":
      "d9be080b33179387e504bb6991ea41ae0dd715e28b01ce9f63d57198a095bccc776874914288e6989e97ac9d255ac667c205fa8d90a211184b417b4ffd

}'
```

Response of `Bob`:

```
{
  "signatures": [
    {
      "id": "009a1f293253e41e",
      "amount": 2,
      "C_": "0224f1c4c564230ad3d96c5033efdc425582397a5a7691d600202732edc6d4b1ec"
    },
    {
      "id": "009a1f293253e41e",
      "amount": 8,
      "C_": "0277d1de806ed177007e5b94a8139343b6382e472c752a74e99949d511f7194f6c"
    }
  ]
}
```

Errors

If the wallet user `Alice` does not include a signature on the `PostMintBolt11Request` but did include a `pubkey` in the `PostMintBolt11QuoteRequest` then `Bob` **MUST** respond with an error. `Alice` **CAN** repeat the request with a valid signature.

See Error Codes:

- `20008`: Mint quote with `pubkey` but no valid `signature` provided for mint request.
- `20009`: Mint quote requires `pubkey` but none given or invalid `pubkey`.

Settings

The settings for this NUT indicate the support for requiring a signature before minting. They are part of the info response of the mint (NUT-06) which in this case reads

```
{
  "20": {
    "supported": <bool>,
```

```
  }
}
```

NUT-21: Clear Authentication

`optional`

`used in: NUT-22`

This NUT defines a clear authentication scheme that allows operators to limit the use of their mint to registered users using the OAuth 2.0 and OpenID Connect protocols. The mint operator can protect chosen endpoints from access by requiring user authentication. Only users that provide a clear authentication token (CAT) from the specified OpenID Connect (OIDC) service can use the protected endpoints. The CAT is an Oauth 2.0 Access token (also known as the `access_token`) commonly in the form of a JWT that contains user information, a signature from the OIDC service, and an expiry time. To access protected endpoints, the wallet includes the CAT in the HTTP request header.

Note: The primary purpose of this NUT is to restrict access to a mint by allowing registered users to obtain Blind Authentication Tokens as specified in NUT-22.

Warning: This authentication scheme breaks the user's privacy as the CAT contains user information. Mint operators SHOULD require clear authentication **only on selected endpoints**, such as those for obtaining blind authentication tokens (BATs, see NUT-22).

OpenID Connect service configuration

The OpenID Connect (OIDC) service is typically run by the mint operator (but it does not have to be). The OIDC service must be configured to meet the following criteria:

- **No client secret:** The OIDC service MUST NOT use a client secret.
- **Authorization code flow:** The OIDC service MUST enable the *authorization code flow* with PKCE for public clients, so that an authorization code can be exchanged for an access token and a refresh token.
- **Signature algorithm:** The OIDC service MUST support at least one of the two asymmetric JWS signature algorithms for access token and ID token signatures: `ES256` and `RS256`.
- **Wallet redirect URLs:** To support the OpenID Connect Authorization Code flow, the OIDC service MUST allow redirect URLs that correspond to the wallets it wants to support. You can find a list of common redirect URLs for well-known Cashu wallets here.
- **Localhost redirect URL:** The OIDC service MUST also allow redirects to the URL `http://localhost:33388/callback`.
- **Authentication flows:** Although, strictly speaking, this NUT does not restrict the OpenID Connect grant types that can be used to obtain a CAT, it is recommended to enable at least the `authorization_code` (Authorization Code) flow and the `urn:ietf:params:oauth:grant-type:device_code` (Device Code) flow in the `grant_types_supported` field of the `openid_discovery` configuration. The `password` (Resource Owner Password Credentials, ROPC) flow SHOULD NOT be used as it requires handling the user's credentials in the wallet application.

Mint

Signalling protected endpoints

The mint lists each protected endpoint that requires a clear authentication token (CAT) in the `MintClearAuthSetting` in its NUT-06 info response:

```
"21" : {
  "openid_discovery": "https://mint.com:8080/realms/nutshell/.well-known/openid-configuration",
  "client_id": "cashu-client",
  "protected_endpoints": [
    {
        "method": "POST",
        "path": "/v1/auth/blind/mint"
    }
  ]
}
```

`openid_discovery` is the OpenID Connect Discovery endpoint which has all the information necessary for a client to authenticate with the service.

`client_id` is the OpenID Connect Client ID that the wallet needs to use to authenticate.

`protected_endpoints` is an array of objects that specify each endpoint that requires a CAT in the request headers. `method` is the HTTP method, and `path` is either:

1. **Exact match**: no trailing * □ request path MUST equal `path`
2. **Prefix match**: ends with * □ request path MUST start with the prefix (* removed)

The * wildcard, if present, MUST be the final character only.

For example:

- `/v1/*` matches any path starting `/v1/` (all endpoints)
- `/v1/auth/*` matches any path that starts with `/v1/auth/` (all auth endpoints)
- `/v1/mint/*` matches any path that starts with `/v1/mint/` (all minting endpoints)
- `/v1/mint/bolt*` matches any path starting `/v1/mint/bolt` (bolt11/12 minting endpoints)

In the example above, the `/v1/auth/blind/mint` path is the **exact match** NUT-22 endpoint for obtaining blind authentication tokens (BATs).

> [!CAUTION] Wallets **MUST** treat mint provided `path` values as untrusted input and use exact or prefix matching only. Never use regex matching on untrusted input.

Clear authentication token verification

When receiving a request to a protected endpoint, the mint checks the included CAT (which is a JWT) in the HTTP request header (see below in section Wallet) and verifies the JWT. To verify the JWT, the mint checks the signature of the OIDC and the expiry of the JWT.

The JWT includes a `sub` field which identifies a specific user. The `sub` identifier can, for example, be used to rate limit the user.

Note: The JWT *MAY* include an *audience* field called `aud` that contains the mint's public key

More on OpenID Connect ID token validation here.

Wallet

To make a request to one of the `protected_endpoints` of the mint, the wallet needs to obtain a valid clear auth token (CAT) from the OIDC service. The wallet uses the `openid_discovery` URL in the `MintClearAuthSetting` from the info endpoint of the mint to authenticate with the OIDC service and obtain a CAT.

Obtaining a CAT

Depending on the wallet implementation and use case, an appropriate authorization flow should be used. For mobile wallets, the Authorization Code is recommended. For command-line wallets, the Device Code flow is recommended. For headless wallets, the ROPC flow may be used.

It is recommended to use language-specific libraries that can handle OpenID Connect authentication on behalf of the user. The wallet should be able to handle and store access tokens and refresh tokens for each mint that it authenticates with. If the wallet connects to a mint for the first time, or if the refresh token is about to expire, the wallet should allow the user to log in again to obtain a new access token (`access_token`) and a new refresh token (`refresh_token`).

The `access_token` is what is referred to as a clear authentication token (CAT) throughout this document.

CAT in request header

When making a request to the mint's endpoint, the wallet matches the requested URL with the `protected_endpoints` from the `MintClearAuthSetting` (either exact match or prefix pattern match). If the match is positive, the mint requires the wallet to provide a CAT with the request.

After obtaining a CAT from the OIDC service, the wallet includes a valid CAT in the HTTP request header when it makes requests to one of the mint's `protected_endpoints`:

```
Clear-auth: <CAT>
```

The `CAT` is a JWT (or `access_token`) encoded with base64 that is signed by and obtained from the OIDC authority. The mint verifies the JWT as described above.

Error codes

See Error Codes:

- `30001`: Endpoint requires clear auth
- `30002`: Clear authentication failed

NUT-22: Blind Authentication

`optional`

`depends on: NUT-21, NUT-12`

This NUT defines a blind authentication scheme that allows mint operators to limit the use of their mint to a set of authorized users while still providing privacy within that anonymity set.

We use two authentication schemes in conjunction: *clear authentication* using an external OpenID Connect / OAuth 2.0 service (described in NUT-21), and *blind authentication* with the mint to access its resources. A user's wallet first needs to obtain a clear authentication token (CAT) from an OpenID Connect authority that the mint selected, which is not the subject of this specification. Once the user has obtained the CAT from the OpenID Connect service, they can use it to obtain multiple blind authentication tokens (BAT) from the mint. We describe this process in this document.

Blind authentication tokens (BATs) are used to access the protected endpoints of the mint and make sure that only users that previously presented a valid CAT can access the mint's features such as minting, melting, or swapping ecash. Wallets provide a BAT in the request header when making a request to one of the mint's protected endpoints. The mint parses the header for a BAT, verifies the signature (like with normal ecash as described in NUT-00), checks if the token has previously been spent, and if not, adds it to its spent BAT token database.

Blind authentication tokens are ecash

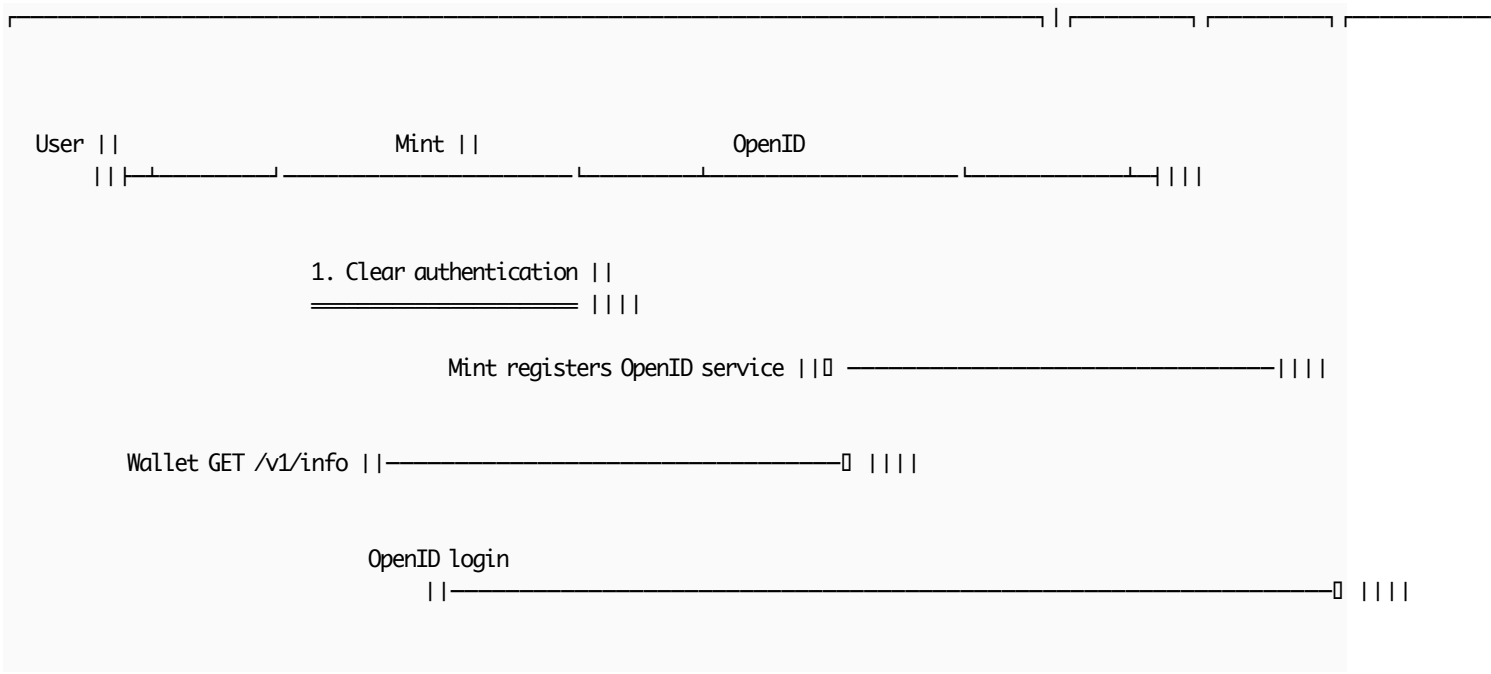
Blind authentication tokens (BATs) are essentially the same as normal ecash tokens and are minted in the same way. They are signed with a special keyset of the mint that has the unit `auth` and a single amount `1`.

BATs can only be used a single time for each request that the wallet makes to the mint's protected endpoints. For each successful request, the BAT is added to the mint's spent token list after which they are regarded as spent. The BAT is not marked as spent if the request results in an error.

To summarize:

- Wallet connects to mint and user is prompted to register or log in with an OAuth 2.0 service
- Upon login, wallet receives a clear authentication (CAT) token that identifies the user
- CAT is used to obtain blind authentication tokens (BAT) from the mint
- BATs are used to access the mint

The diagram below illustrates the protocol flow.



The steps for `1. Clear authentication` are described in NUT-21, whereas the steps in `2. Blind authentication` are subject of this document.

Endpoints

The mint offers new endpoints that behave similarly to the endpoints for getting the keys, keysets, and minting tokens with normal ecash (NUT-01, NUT-02, NUT-04). These endpoints start with a prefix `/v1/auth/` to differentiate them from the normal endpoints of the mint. Using these endpoints, wallets can mint blind authentication tokens (BATs) and use them later when accessing the protected endpoints of the mint. Note that BATs cannot be swapped against other BATs.

Keys

Like in NUT-01 and NUT-02, the mint responds with its BAT keyset for the following request:

```
GET /v1/auth/blind/keys
```

or

```
GET /v1/auth/blind/keys/{keyset_id}
```

where the mint returns a `GetKeysResponse`:

```
{
  "keysets": [
    {
      "id": "000e479673849bf6",
      "unit": "auth",
      "keys": {
        "1": "024ec000e31e230e4c59760def29601557c0b1650617dc8f38d3b2cfd21ad0351b"
      }
    }
  ]
}
```

Notice that the unit is `auth` and only a single amount of `1` is supported.

Keysets

Like in NUT-02 the mint also offers the endpoints returning the keysets:

```
GET v1/auth/blind/keysets
```

The mint returns the same `GetKeysetsResponse` response types as described in NUT-02.

Minting blind authentication tokens

To mint blind authentication tokens (BATs), the wallet makes a request to the following endpoint:

```
POST /v1/auth/blind/mint
```

To access this endpoint the wallet MUST provide a valid CAT (obtained via NUT-21) in its request header, IF this endpoint is marked as protected in the info response of the mint as per NUT-21.

```
Clear-auth: <CAT>
```

Like in NUT-04, the wallet includes a `PostAuthBlindMintRequest` in the request body:

```
{
  "outputs": <Array[BlindedMessage]>
}
```

where `outputs` are `BlindedMessages` (see NUT-00) from the blind auth keyset of the mint with a unit `amount`. The sum of all amounts of the outputs cannot exceed the maximum allowed amount of BATs as specified in `bat_max_mint` in the mint's `MintBlindAuthSetting`.

Notice that in contrast to NUT-04, we did not create a quote and did not include it in this request. Instead, we directly minted the maximum allowed amount of BATs.

The mint responds with a `PostAuthBlindMintResponse`:

```
{
  "signatures": <Array[BlindSignature]>
}
```

The wallet un-blinds the response to obtain the signatures `C` as described in NUT-00. It then stores the resulting `AuthProofs` in its database:

```
{
  "id": <hex_str>,
  "secret": <str>,
  "C": <hex_str>,
  "dleq": {
    "e": <str>,
    "s": <str>,
    "r": <str>
  }
}
```

To prevent pinning, wallets MUST validate the DLEQ proofs `dleq` as defined in NUT-12. Should `AuthProofs` be sent to another user of the mint, it MUST include the `dleq` proof so that the receiving user can validate it.

Using blind authentication tokens

The wallet checks the `MintBlindAuthSetting` of the mint to determine which endpoints require blind authentication. Similar to `NUT-21`, the wallet performs a match on the `protected_endpoints` in the `MintBlindAuthSetting` before attempting a request to one of the mint's endpoints. If the match is positive, the wallet needs to add a blind authentication token (BAT) to the request header.

Serialization

To add a blind authentication token (BAT) to the request header, we need to serialize a single `AuthProof` JSON in base64 with the prefix `authA`:

```
authA[base64_authproof_json]
```

This string is a BAT.

[!CAUTION]

To protect the privacy of the wallet, the BAT MUST NOT contain the `dleq` proof when it is sent to the mint in the request header.

Request header

We add this serialized BAT to the request header:

```
Blind-auth: <BAT>
```

and make the request as we usually would.

`AuthProofs` are single-use. The wallet MUST delete the `AuthProof` after a successful request, and SHOULD delete it even if request results in an error. If the wallet runs out of `AuthProofs`, it can mint new ones using its clear authentication token (CAT).

Mint

DLEQs

The mint MUST return DLEQ proofs for every blind signature in `PostAuthBlindMintResponse` as defined in NUT-12

Signaling protected endpoints and settings

The mint lists each protected endpoint that requires a blind authentication token (BAT) in the `MintBlindAuthSetting` in its NUT-06 info response:

```
"22" : {
  "bat_max_mint": 50,
  "protected_endpoints": [
    {
      "method": "GET",
      "path": "/v1/mint/*"
    },
    {
      "method": "POST",
      "path": "/v1/mint/*"
    }
  ]
}
```

`bat_max_mint` is the number of blind authentication tokens (BATs) that can be minted in a single request using the `POST /v1/auth/blind/mint` endpoint.

`protected_endpoints` contains the endpoints that are protected by blind authentication. `method` denotes the HTTP method of the endpoint, and `path` is either:

1. **Exact match**: no trailing * □ request path MUST equal `path`
2. **Prefix match**: ends with * □ request path MUST start with the prefix (* removed)

The * wildcard, if present, MUST be the final character only.

For example:

- `/v1/*` matches any path starting `/v1/` (all endpoints)
- `/v1/mint/*` matches any path that starts with `/v1/mint/` (all minting endpoints)
- `/v1/mint/bolt*` matches any path starting `/v1/mint/bolt` (bolt11/12 minting endpoints)

  [!CAUTION] Wallets **MUST** treat mint provided `path` values as untrusted input and use exact or prefix matching only. Never use regex matching on untrusted input.

Error codes

See Error Codes:

- `31001`: Endpoint requires blind auth
- `31002`: Blind authentication failed
- `31003`: Maximum BAT mint amount exceeded
- `31004`: BAT mint rate limit exceeded

NUT-23: BOLT11

`optional`

`depends on: NUT-04 NUT-05`

This document describes minting and melting ecash with the `bolt11` payment method, which uses Lightning Network invoices. It is an extension of NUT-04 and NUT-05 which cover the protocol steps of minting and melting ecash shared by any supported payment method.

Mint Quote

For the `bolt11` method, the wallet includes the following specific `PostMintQuoteBolt11Request` data:

```
{
  "amount": <int>,
  "unit": <str_enum[UNIT]>,
  "description": <str> // Optional
}
```

The mint responds with a `PostMintQuoteBolt11Response`:

```
{
  "quote": <str>,
  "request": <str>, // The bolt11 invoice to pay
  "amount": <int>,
  "unit":  <str_enum[UNIT]>,
  "state": <str_enum[STATE]>,
  "expiry": <int|null>
}
```

`state` is an enum string field with possible values `"UNPAID"`, `"PAID"`, `"ISSUED"`:

- `"UNPAID"` means that the quote's request has not been paid yet.
- `"PAID"` means that the quote's request has been paid but the ecash is not issued yet.
- `"ISSUED"` means that the quote has been paid and the ecash has been issued.

`expiry` is the Unix timestamp until which the `request` can be paid (i.e. the bolt11 invoice expiry).

Example

Request with curl:

```
curl -X POST http://localhost:3338/v1/mint/quote/bolt11 -d '{"amount": 10, "unit": "sat"}' -H "Content-Type:
    application/json"
```

Response:

```
{
  "quote": "DSGLX9kevM...",
  "request": "lnbc100n1pj4apw9...",
  "amount": 10,
  "unit": "sat",
  "state": "UNPAID",
  "expiry": 1701704757
}
```

Check quote state:

```
curl -X GET http://localhost:3338/v1/mint/quote/bolt11/DSGLX9kevM...
```

Minting tokens:

```
curl -X POST https://mint.host:3338/v1/mint/bolt11 -H "Content-Type: application/json" -d \
'{
  "quote": "DSGLX9kevM...",
  "outputs": [
    {
      "amount": 8,
      "id": "009a1f293253e41e",
      "B_": "035015e6d7ade60ba8426cefaf1832bbd27257636e44a76b922d78e79b47cb689d"
    },
    {
      "amount": 2,
      "id": "009a1f293253e41e",
      "B_": "0288d7649652d0a83fc9c966c969fb217f15904431e61a44b14999fabc1b5d9ac6"
    }
  ]
}'
```

Response:

```
{
  "signatures": [
    {
      "id": "009a1f293253e41e",
      "amount": 2,
      "C_": "0224f1c4c564230ad3d96c5033efdc425582397a5a7691d600202732edc6d4b1ec"
    },
    {
      "id": "009a1f293253e41e",
      "amount": 8,
      "C_": "0277d1de806ed177007e5b94a8139343b6382e472c752a74e99949d511f7194f6c"
    }
  ]
}
```

Mint Settings

A `description` option **MUST** be set to indicate whether the `bolt11` payment method backend supports providing an invoice description.

Example `MintMethodSetting`

```
{
  "method": "bolt11",
  "unit": "sat",
  "min_amount": 0,
  "max_amount": 10000,
  "options": {
    "description": true
  }
}
```

Melt Quote

For the `bolt11` method, the wallet includes the following specific `PostMeltQuoteBolt11Request` data:

```
{
  "request": <str>,
  "unit": <str_enum[UNIT]>,
  "options": { // Optional
    "amountless": {
      "amount_msat": <int>
    }
  }
}
```

Here, `request` is the bolt11 Lightning invoice to be paid and `unit` is the unit the wallet would like to pay with. The `options` field can include support for amountless invoices if supported by the mint.

The mint responds with a `PostMeltQuoteBolt11Response`:

```
{
  "quote": <str>,
  "request": <str>,
  "amount": <int>,
  "unit": <str_enum[UNIT]>,
  "fee_reserve": <int>,
  "state": <str_enum[STATE]>,
  "expiry": <int>,
  "payment_preimage": <str|null>
}
```

Where `fee_reserve` is the additional fee reserve required for the Lightning payment. The mint expects the wallet to include `Proofs` of *at least* `total_amount = amount + fee_reserve + fee` where `fee` is calculated from the keyset's `input_fee_ppk` as described in NUT-02.

Melting Tokens

For the `bolt11` method, the wallet can include an optional `outputs` field in the melt request to receive change for overpaid Lightning fees (see NUT-08):

```
{
  "quote": <str>,
  "inputs": <Array[Proof]>,
  "outputs": <Array[BlindedMessage]> // Optional
}
```

If the `outputs` field is included and there is excess from the `fee_reserve`, the mint will respond with a `change` field containing blind signatures for the overpaid amount:

```
{
  "quote": <str>,
  "request": <str>,
  "amount": <int>,
  "unit": <str_enum[UNIT]>,
  "fee_reserve": <int>,
  "state": <str_enum[STATE]>,
  "expiry": <int>,
  "payment_preimage": <str>,
  "change": <Array[BlindSignature]> // Present if outputs were included and there's change
}
```

Example

Melt quote request:

```
curl -X POST https://mint.host:3338/v1/melt/quote/bolt11 -d \
'{"request": "lnbc100n1p3kdrv5sp5lpdxzghe5j67q...", "unit": "sat"}'
```

Melt quote response:

```
{
  "quote": "TRmjduhIsPxd...",
  "request": "lnbc100n1p3kdrv5sp5lpdxzghe5j67q...",
  "amount": 10,
  "unit": "sat",
  "fee_reserve": 2,
  "state": "UNPAID",
  "expiry": 1701704757
}
```

Check quote state:

```
curl -X GET http://localhost:3338/v1/melt/quote/bolt11/TRmjduhIsPxd...
```

Melt request:

```
curl -X POST https://mint.host:3338/v1/melt/bolt11 -d \
'{"quote": "od4CN5smMMS3K3QVHkbGGNCTxfcAIyIXeq8IrfhP", "inputs": [...]}'
```

Successful melt response:

```
{
  "quote": "TRmjduhIsPxd...",
  "request": "lnbc100n1p3kdrv5sp5lpdxzghe5j67q...",
  "amount": 10,
  "unit": "sat",
  "fee_reserve": 2,
  "state": "PAID",
  "expiry": 1701704757,
  "payment_preimage": "c5a1ae1f639e1f4a3872e81500fd028bece7bedc1152f740cba5c3417b748c1b"
}
```

Example `MeltMethodSetting`

```
{
  "method": "bolt11",
  "unit": "sat",
  "min_amount": 100,
  "max_amount": 10000,
  "options": {
    "amountless": true
  }
}
```

NUT-24: HTTP 402 Payment Required

`optional`

`uses: NUT-12`

`depends on: NUT-18`

This NUT describes how Cashu tokens can be used with HTTP 402 Payment Required responses to enable payments for HTTP resources.

Server response

HTTP servers may respond with the HTTP status code 402 with a `X-Cashu` header containing an encoded NUT-18 payment request with the following fields:

```
{
    "a": int,
    "u": str,
    "m": Array[str],
    "nut10": NUT10Option <optional>
}
```

- `a`: The amount required in the specified unit
- `u`: The currency unit (e.g., "sat", "usd", "api")
- `m`: Array of mint URLs that the server accepts tokens from
- `nut10`: The required NUT-10 locking condition

Note that there is no transport field since we expect an in-band payment.

Client payment

After receiving a 402 response with a `X-Cashu` header, the client may retry the request with a `cashuB` token in the `X-Cashu` header as payment.

The token MUST be from one of the mints listed in the `mints` array, and MUST be of the same `unit` and greater than or equal to the `amount`.

Payment response

If the server receives tokens from a mint that is not in the `mints` array, an incorrect `unit`, an insufficient amount of tokens, or with insufficient locking conditions, it should respond with a HTTP `400` status code.

NUT-25: BOLT12

`optional`

`depends on: NUT-04 NUT-05 NUT-20`

This document describes minting and melting ecash with the `bolt12` payment method, which uses Lightning Network offers. It is an extension of NUT-04 and NUT-05 which cover the protocol steps of minting and melting ecash shared by any supported payment method.

Mint Quote

For the `bolt12` method, the wallet includes the following specific `PostMintQuoteBolt12Request` data:

```
{
  "amount": <int|null>,
  "unit": <str_enum[UNIT]>,
  "description": <str|null>,
  "pubkey": <str>
}
```

> **Note:** While a pubkey is optional as per NUT-20 for NUT-04 it is required in this NUT and the mint **MUST NOT** issue a mint quote if one is not included.
>
> **Privacy:** To prevent linking multiple mint quotes together, wallets **SHOULD** generate a unique public key for each mint quote request.

The mint responds with a `PostMintQuoteBolt12Response`:

```
{
  "quote": <str>,
  "request": <str>,
  "amount": <int|null>,
  "unit": <str_enum[UNIT]>,
  "expiry": <int|null>,
  "pubkey": <str>,
  "amount_paid": <int>,
  "amount_issued": <int>
}
```

Where:

- `quote` is the quote ID
- `request` is the bolt12 offer
- `expiry` is the Unix timestamp until which the mint quote is valid
- `amount_paid` is the amount that has been paid to the mint via the bolt12 offer
- `amount_issued` is the amount of ecash that has been issued for the given mint quote

Example

Request with curl:

```
curl -X POST http://localhost:3338/v1/mint/quote/bolt12 -d \
'{"amount": 10, "unit": "sat", "pubkey": "03d56ce4e446a85bbdaa547b4ec2b073d40ff802831352b8272b7dd7a4de5a7cac"}' \
-H "Content-Type: application/json"
```

Response:

```
{
  "quote": "DSGLX9kevM...",
```

```
  "request": "lno1qcp...",
  "amount": 10,
  "unit": "sat",
  "expiry": 1701704757,
  "pubkey": "03d56ce4e446a85bbdaa547b4ec2b073d40ff802831352b8272b7dd7a4de5a7cac",
  "amount_paid": 0,
  "amount_issued": 0
}
```

Check quote state:

```
curl -X GET http://localhost:3338/v1/mint/quote/bolt12/DSGLX9kevM...
```

Minting tokens:

```
curl -X POST https://mint.host:3338/v1/mint/bolt12 -H "Content-Type: application/json" -d \
'{
  "quote": "DSGLX9kevM...",
  "outputs": [
    {
      "amount": 8,
      "id": "009a1f293253e41e",
      "B_": "035015e6d7ade60ba8426cefaf1832bbd27257636e44a76b922d78e79b47cb689d"
    },
    {
      "amount": 2,
      "id": "009a1f293253e41e",
      "B_": "0288d7649652d0a83fc9c966c969fb217f15904431e61a44b14999fabc1b5d9ac6"
    }
  ]
}'
```

Response:

```
{
  "signatures": [
    {
      "id": "009a1f293253e41e",
      "amount": 2,
      "C_": "0224f1c4c564230ad3d96c5033efdc425582397a5a7691d600202732edc6d4b1ec"
    },
    {
      "id": "009a1f293253e41e",
      "amount": 8,
      "C_": "0277d1de806ed177007e5b94a8139343b6382e472c752a74e99949d511f7194f6c"
    }
  ]
}
```

Multiple Issuances

Unlike BOLT11 invoices, BOLT12 offers can be paid multiple times, allowing the wallet to mint multiple times for one quote. The wallet can call the check bolt12 endpoint, where the mint will return the `PostMintQuoteBolt12Response` including `amount_paid` and `amount_issued`. The difference between these values represents how much the wallet can mint by calling the mint endpoint. Wallets MAY mint any amount up to this available difference; in particular, they can mint less than the amount mintable. Mints MUST accept mint requests whose total output amount is less than or equal to (`amount_paid` - `amount_issued`).

Mint Settings

A `description` option **SHOULD** be set to indicate whether the `bolt12` payment method backend supports providing an offer description.

Example `MintMethodSetting`

```
{
  "method": "bolt12",
  "unit": <str>,
  "min_amount": <int|null>,
  "max_amount": <int|null>,
  "options": {
    "description": true
  }
}
```

Melt Quote

For the `bolt12` method, the wallet includes the following specific `PostMeltQuoteBolt12Request` data:

```
{
  "request": <str>,
  "unit": <str_enum[UNIT]>,
  "options": { // Optional
    "amountless": {
      "amount_msat": <int>
    }
  }
}
```

Here, `request` is the bolt12 Offer to be paid and `unit` is the unit the wallet would like to pay with. For amount-less offers, the `options.amountless.amount_msat` field can be used to specify the amount in millisatoshis to pay to the offer. If `options.amountless.amount_msat` is defined and the offer has an amount, they **MUST** be equal.

The mint responds with a `PostMeltQuoteBolt12Response`:

```
{
  "quote": <str>,
  "request": <str>,
  "amount": <int>,
  "unit": <str_enum[UNIT]>,
  "fee_reserve": <int>,
  "state": <str_enum[STATE]>,
  "expiry": <int>,
  "payment_preimage": <str|null>
}
```

Where `fee_reserve` is the additional fee reserve required for the Lightning payment. The mint expects the wallet to include `Proofs` of *at least* `total_amount = amount + fee_reserve + fee` where `fee` is calculated from the keyset's `input_fee_ppk` as described in NUT-02.

`state` is an enum string field with possible values `"UNPAID"`, `"PENDING"`, `"PAID"`:

- `"UNPAID"` means that the request has not been paid yet.
- `"PENDING"` means that the request is currently being paid.
- `"PAID"` means that the request has been paid successfully.

Melting Tokens

For the `bolt12` method, the wallet can include an optional `outputs` field in the melt request to receive change for overpaid Lightning fees (see NUT-08):

```
{
  "quote": <str>,
  "inputs": <Array[Proof]>,
  "outputs": <Array[BlindedMessage]> // Optional
}
```

If the `outputs` field is included and there is excess from the `fee_reserve`, the mint will respond with a `change` field containing blind signatures for the overpaid amount:

```
{
  "quote": <str>,
  "request": <str>,
  "amount": <int>,
  "unit": <str_enum[UNIT]>,
  "fee_reserve": <int>,
  "state": <str_enum[STATE]>,
  "expiry": <int>,
  "payment_preimage": <str>,
  "change": <Array[BlindSignature]> // Present if outputs were included and there's change
}
```

Example

Melt quote request:

```
curl -X POST https://mint.host:3338/v1/melt/quote/bolt12 -d \
'{"request": "lno1qcp4256ypqpq86q69t5wv5629arxqurn8cxg9p5qmmqy2e5xq...", "unit": "sat"}'
```

Melt quote response:

```
{
  "quote": "TRmjduhIsPxd...",
  "request": "lno1qcp4256ypqpq86q69t5wv5629arxqurn8cxg9p5qmmqy2e5xq...",
  "amount": 10,
  "unit": "sat",
  "fee_reserve": 2,
  "state": "UNPAID",
  "expiry": 1701704757
}
```

Check quote state:

```
curl -X GET http://localhost:3338/v1/melt/quote/bolt12/TRmjduhIsPxd...
```

Melt request:

```
curl -X POST https://mint.host:3338/v1/melt/bolt12 -d \
'{"quote": "TRmjduhIsPxd...", "inputs": [...]}'
```

Successful melt response:

```
{
  "quote": "TRmjduhIsPxd...",
  "request": "lno1qcp4256ypqpq86q69t5wv5629arxqurn8cxg9p5qmmqy2e5xq...",
  "amount": 10,
  "unit": "sat",
```

```
  "fee_reserve": 2,
  "state": "PAID",
  "expiry": 1701704757,
  "payment_preimage": "c5a1ae1f639e1f4a3872e81500fd028bece7bedc1152f740cba5c3417b748c1b"
}
```

Example `MeltMethodSetting`

```
{
  "method": "bolt12",
  "unit": <str>,
  "min_amount": <int|null>,
  "max_amount": <int|null>
}
```

Conclusion

Thank you for exploring the Cashu Book of NUTs. My hope is that this reorganized collection of Cashu Book of (NUTs) has provided you with a clearer and more structured understanding of the Cashu protocol. The aim was to make the information more accessible to everyone.

As we wrap up this book, remember that the journey with Cashu doesn't end here. The protocol is continuously evolving, and your engagement and contributions are crucial for its growth and refinement. I encourage you to participate in the discussions, contribute your ideas, and help in developing this open and decentralized platform.

Once again, all the credit for the content in this book goes to the original authors of the NUTs. This compilation is merely a tool to assist in navigating their innovative work. Whether you're a developer, researcher, or enthusiast, your insights and enthusiasm are what will propel Cashu forward.

Let's keep the spirit of innovation and collaboration alive. Here's to building a more connected and decentralized future together!