www.e2encrypted.com /nostr/nips/

# Nostr protocol in a single page

12/02/2013

Last Updated: 2025-01-01

nostr

- nostr
- protocol
- privacy

Page content

## NIPs

NIPs stand for **Nostr Implementation Possibilities**.

They exist to document what may be implemented by Nostr -compatible *relay* and *client* software.

---

### List

- NIP-01: Basic protocol flow description
- NIP-02: Follow List
- NIP-03: OpenTimestamps Attestations for Events
- NIP-04: Encrypted Direct Message — **unrecommended**: deprecated in favor of NIP-17
- NIP-05: Mapping Nostr keys to DNS-based internet identifiers
- NIP-06: Basic key derivation from mnemonic seed phrase
- NIP-07: `window.nostr` capability for web browsers
- NIP-08: Handling Mentions — **unrecommended**: deprecated in favor of NIP-27
- NIP-09: Event Deletion Request
- NIP-10: Conventions for clients' use of `e` and `p` tags in text events
- NIP-11: Relay Information Document
- NIP-13: Proof of Work
- NIP-14: Subject tag in text events
- NIP-15: Nostr Marketplace (for resilient marketplaces)
- NIP-17: Private Direct Messages
- NIP-18: Reposts
- NIP-19: bech32-encoded entities
- NIP-21: `nostr:` URI scheme
- NIP-22: Comment
- NIP-23: Long-form Content
- NIP-24: Extra metadata fields and tags
- NIP-25: Reactions
- NIP-26: Delegated Event Signing
- NIP-27: Text Note References
- NIP-28: Public Chat
- NIP-29: Relay-based Groups
- NIP-30: Custom Emoji
- NIP-31: Dealing with Unknown Events
- NIP-32: Labeling
- NIP-34: `git` stuff
- NIP-35: Torrents
- NIP-36: Sensitive Content
- NIP-37: Draft Events
- NIP-38: User Statuses
- NIP-39: External Identities in Profiles
- NIP-40: Expiration Timestamp
- NIP-42: Authentication of clients to relays
- NIP-44: Encrypted Payloads (Versioned)
- NIP-45: Counting results
- NIP-46: Nostr Remote Signing

- NIP-47: Nostr Wallet Connect
- NIP-48: Proxy Tags
- NIP-49: Private Key Encryption
- NIP-50: Search Capability
- NIP-51: Lists
- NIP-52: Calendar Events
- NIP-53: Live Activities
- NIP-54: Wiki
- NIP-55: Android Signer Application
- NIP-56: Reporting
- NIP-57: Lightning Zaps
- NIP-58: Badges
- NIP-59: Gift Wrap
- NIP-60: Cashu Wallet
- NIP-61: Nutzaps
- NIP-64: Chess (PGN)
- NIP-65: Relay List Metadata
- NIP-68: Picture-first feeds
- NIP-69: Peer-to-peer Order events
- NIP-70: Protected Events
- NIP-71: Video Events
- NIP-72: Moderated Communities
- NIP-73: External Content IDs
- NIP-75: Zap Goals
- NIP-78: Application-specific data
- NIP-84: Highlights
- NIP-86: Relay Management API
- NIP-89: Recommended Application Handlers
- NIP-90: Data Vending Machines
- NIP-92: Media Attachments
- NIP-94: File Metadata
- NIP-96: HTTP File Storage Integration
- NIP-98: HTTP Auth
- NIP-99: Classified Listings
- NIP-7D: Threads
- NIP-C7: Chats

### Event Kinds

| kind | description | NIP |
|---|---|---|
| 0 | User Metadata | 01 |
| 1 | Short Text Note | 01 |
| 2 | Recommend Relay | 01 (deprecated) |
| 3 | Follows | 02 |
| 4 | Encrypted Direct Messages | 04 |
| 5 | Event Deletion Request | 09 |
| 6 | Repost | 18 |
| 7 | Reaction | 25 |
| 8 | Badge Award | 58 |
| 9 | Chat Message | C7 |
| 10 | Group Chat Threaded Reply | 29 (deprecated) |
| 11 | Thread | 7D |
| 12 | Group Thread Reply | 29 (deprecated) |
| 13 | Seal | 59 |
| 14 | Direct Message | 17 |
| 16 | Generic Repost | 18 |
| 17 | Reaction to a website | 25 |
| 20 | Picture | 68 |
| 40 | Channel Creation | 28 |
| 41 | Channel Metadata | 28 |
| 42 | Channel Message | 28 |
| 43 | Channel Hide Message | 28 |
| 44 | Channel Mute User | 28 |
| 64 | Chess (PGN) | 64 |
| 818 | Merge Requests | 54 |
| 1021 | Bid | 15 |
| 1022 | Bid confirmation | 15 |
| 1040 | OpenTimestamps | 03 |

| kind | description | NIP |
|---|---|---|
| 1059 | Gift Wrap | 59 |
| 1063 | File Metadata | 94 |
| 1111 | Comment | 22 |
| 1311 | Live Chat Message | 53 |
| 1617 | Patches | 34 |
| 1621 | Issues | 34 |
| 1622 | Replies | 34 |
| 1630-1633 | Status | 34 |
| 1971 | Problem Tracker | nostrocket |
| 1984 | Reporting | 56 |
| 1985 | Label | 32 |
| 1986 | Relay reviews | |
| 1987 | AI Embeddings / Vector lists | NKBIP-02 |
| 2003 | Torrent | 35 |
| 2004 | Torrent Comment | 35 |
| 2022 | Coinjoin Pool | joinstr |
| 4550 | Community Post Approval | 72 |
| 5000-5999 | Job Request | 90 |
| 6000-6999 | Job Result | 90 |
| 7000 | Job Feedback | 90 |
| 7374 | Reserved Cashu Wallet Tokens | 60 |
| 7375 | Cashu Wallet Tokens | 60 |
| 7376 | Cashu Wallet History | 60 |
| 9000-9030 | Group Control Events | 29 |
| 9041 | Zap Goal | 75 |
| 9321 | Nutzap | 61 |
| 9467 | Tidal login | Tidal-nostr |
| 9734 | Zap Request | 57 |
| 9735 | Zap | 57 |
| 9802 | Highlights | 84 |
| 10000 | Mute list | 51 |
| 10001 | Pin list | 51 |
| 10002 | Relay List Metadata | 65 |
| 10003 | Bookmark list | 51 |
| 10004 | Communities list | 51 |
| 10005 | Public chats list | 51 |
| 10006 | Blocked relays list | 51 |
| 10007 | Search relays list | 51 |
| 10009 | User groups | 51 , 29 |
| 10013 | Draft relays | 37 |
| 10015 | Interests list | 51 |
| 10019 | Nutzap Mint Recommendation | 61 |
| 10030 | User emoji list | 51 |
| 10050 | Relay list to receive DMs | 51 , 17 |
| 10063 | User server list | Blossom |
| 10096 | File storage server list | 96 |
| 13194 | Wallet Info | 47 |
| 21000 | Lightning Pub RPC | Lightning.Pub |
| 22242 | Client Authentication | 42 |
| 23194 | Wallet Request | 47 |
| 23195 | Wallet Response | 47 |
| 24133 | Nostr Connect | 46 |
| 24242 | Blobs stored on mediaservers | Blossom |
| 27235 | HTTP Auth | 98 |
| 30000 | Follow sets | 51 |
| 30001 | Generic lists | 51 (deprecated) |
| 30002 | Relay sets | 51 |
| 30003 | Bookmark sets | 51 |
| 30004 | Curation sets | 51 |
| 30005 | Video sets | 51 |
| 30007 | Kind mute sets | 51 |
| 30008 | Profile Badges | 58 |
| 30009 | Badge Definition | 58 |
| 30015 | Interest sets | 51 |
| 30017 | Create or update a stall | 15 |
| 30018 | Create or update a product | 15 |
| 30019 | Marketplace UI/UX | 15 |

| kind | description | NIP |
|---|---|---|
| 30020 | Product sold as an auction | 15 |
| 30023 | Long-form Content | 23 |
| 30024 | Draft Long-form Content | 23 |
| 30030 | Emoji sets | 51 |
| 30040 | Modular Article Header | NKBIP-01 |
| 30041 | Modular Article Content | NKBIP-01 |
| 30063 | Release artifact sets | 51 |
| 30078 | Application-specific Data | 78 |
| 30311 | Live Event | 53 |
| 30315 | User Statuses | 38 |
| 30388 | Slide Set | Corny Chat |
| 30402 | Classified Listing | 99 |
| 30403 | Draft Classified Listing | 99 |
| 30617 | Repository announcements | 34 |
| 30618 | Repository state announcements | 34 |
| 30818 | Wiki article | 54 |
| 30819 | Redirects | 54 |
| 31234 | Draft Event | 37 |
| 31388 | Link Set | Corny Chat |
| 31890 | Feed | NUD: Custom Feeds |
| 31922 | Date-Based Calendar Event | 52 |
| 31923 | Time-Based Calendar Event | 52 |
| 31924 | Calendar | 52 |
| 31925 | Calendar Event RSVP | 52 |
| 31989 | Handler recommendation | 89 |
| 31990 | Handler information | 89 |
| 34235 | Video Event | 71 |
| 34236 | Short-form Portrait Video Event | 71 |
| 34550 | Community Definition | 72 |
| 37375 | Cashu Wallet Event | 60 |
| 38383 | Peer-to-peer Order events | 69 |
| 39000-9 | Group metadata events | 29 |

## Message types

### Client to Relay

| type | description | NIP |
|---|---|---|
| EVENT | used to publish events | 01 |
| REQ | used to request events and subscribe to new updates | 01 |
| CLOSE | used to stop previous subscriptions | 01 |
| AUTH | used to send authentication events | 42 |
| COUNT | used to request event counts | 45 |

### Relay to Client

| type | description | NIP |
|---|---|---|
| EOSE | used to notify clients all stored events have been sent | 01 |
| EVENT | used to send events requested to clients | 01 |
| NOTICE | used to send human-readable messages to clients | 01 |
| OK | used to notify clients if an EVENT was successful | 01 |
| CLOSED | used to notify clients that a REQ was ended and why | 01 |
| AUTH | used to send authentication challenges | 42 |
| COUNT | used to send requested event counts to clients | 45 |

## Standardized Tags

| name | value | other parameters | NIP |
|---|---|---|---|
| a | coordinates to an event | relay URL | 01 |
| A | root address | relay URL | 22 |
| d | identifier | – | 01 |
| e | event id (hex) | relay URL, marker, pubkey (hex) | 01 , 10 |
| E | root event id | relay URL | 22 |
| f | currency code | – | 69 |
| g | geohash | – | 52 |
| h | group id | – | 29 |
| i | external identity | proof, url hint | 35 , 39 , 73 |
| I | root external identity | – | 22 |

| name | value | other parameters | NIP |
|---|---|---|---|
| `k` | kind | – | 18 , 25 , 72 , 73 |
| `K` | root scope | – | 22 |
| `l` | label, label namespace | – | 32 |
| `L` | label namespace | – | 32 |
| `m` | MIME type | – | 94 |
| `p` | pubkey (hex) | relay URL, petname | 01 , 02 |
| `P` | pubkey (hex) | – | 57 |
| `q` | event id (hex) | relay URL, pubkey (hex) | 18 |
| `r` | a reference (URL, etc) | – | 24 , 25 |
| `r` | relay url | marker | 65 |
| `s` | status | – | 69 |
| `t` | hashtag | – | 24 , 34 , 35 |
| `u` | url | – | 61 , 98 |
| `x` | infohash | – | 35 |
| `y` | platform | – | 69 |
| `z` | order number | – | 69 |
| `-` | – | – | 70 |
| `alt` | summary | – | 31 |
| `amount` | millisatoshis, stringified | – | 57 |
| `bolt11` | `bolt11` invoice | – | 57 |
| `challenge` | challenge string | – | 42 |
| `client` | name, address | relay URL | 89 |
| `clone` | git clone URL | – | 34 |
| `content-warning` | reason | – | 36 |
| `delegation` | pubkey, conditions, delegation token | – | 26 |
| `description` | description | – | 34 , 57 , 58 |
| `emoji` | shortcode, image URL | – | 30 |
| `encrypted` | – | – | 90 |
| `expiration` | unix timestamp (string) | – | 40 |
| `file` | full path (string) | – | 35 |
| `goal` | event id (hex) | relay URL | 75 |
| `image` | image URL | dimensions in pixels | 23 , 52 , 58 |
| `imeta` | inline metadata | – | 92 |
| `lnurl` | `bech32` encoded `lnurl` | – | 57 |
| `location` | location string | – | 52 , 99 |
| `name` | name | – | 34 , 58 , 72 |
| `nonce` | random | difficulty | 13 |
| `preimage` | hash of `bolt11` invoice | – | 57 |
| `price` | price | currency, frequency | 99 |
| `proxy` | external ID | protocol | 48 |
| `published_at` | unix timestamp (string) | – | 23 |
| `relay` | relay url | – | 42 , 17 |
| `relays` | relay list | – | 57 |
| `server` | file storage server url | – | 96 |
| `subject` | subject | – | 14 , 17 , 34 |
| `summary` | summary | – | 23 , 52 |
| `thumb` | badge thumbnail | dimensions in pixels | 58 |
| `title` | article title | – | 23 |
| `tracker` | torrent tracker URL | – | 35 |
| `web` | webpage URL | – | 34 |
| `zap` | pubkey (hex), relay URL | weight | 57 |

Please update these lists when proposing new NIPs.

### Criteria for acceptance of NIPs

1. They should be fully implemented in at least two clients and one relay – when applicable.
2. They should make sense.
3. They should be optional and backwards-compatible: care must be taken such that clients and relays that choose to not implement them do not stop working when interacting with the ones that choose to.
4. There should be no more than one way of doing the same thing.
5. Other rules will be made up when necessary.

### Is this repository a centralizing factor?

To promote interoperability, we need standards that everybody can follow, and we need them to define a **single way of doing each thing** without ever hurting **backwards-compatibility**, and for that purpose there is no way around getting everybody to agree on the same thing and keep a centralized index of these standards. However the fact that

such an index exists doesn't hurt the decentralization of Nostr. *At any point the central index can be challenged if it is failing to fulfill the needs of the protocol* and it can migrate to other places and be maintained by other people.

It can even fork into multiple versions, and then some clients would go one way, others would go another way, and some clients would adhere to both competing standards. This would hurt the simplicity, openness and interoperability of Nostr a little, but everything would still work in the short term.

There is a list of notable Nostr software developers who have commit access to this repository, but that exists mostly for practical reasons, as by the nature of the thing we're dealing with the repository owner can revoke membership and rewrite history as they want – and if these actions are unjustified or perceived as bad or evil the community must react.

### How this repository works

Standards may emerge in two ways: the first way is that someone starts doing something, then others copy it; the second way is that someone has an idea of a new standard that could benefit multiple clients and the protocol in general without breaking **backwards-compatibility** and the principle of having **a single way of doing things**, then they write that idea and submit it to this repository, other interested parties read it and give their feedback, then once most people reasonably agree we codify that in a NIP which client and relay developers that are interested in the feature can proceed to implement.

These two ways of standardizing things are supported by this repository. Although the second is preferred, an effort will be made to codify standards emerged outside this repository into NIPs that can be later referenced and easily understood and implemented by others – but obviously as in any human system discretion may be applied when standards are considered harmful.

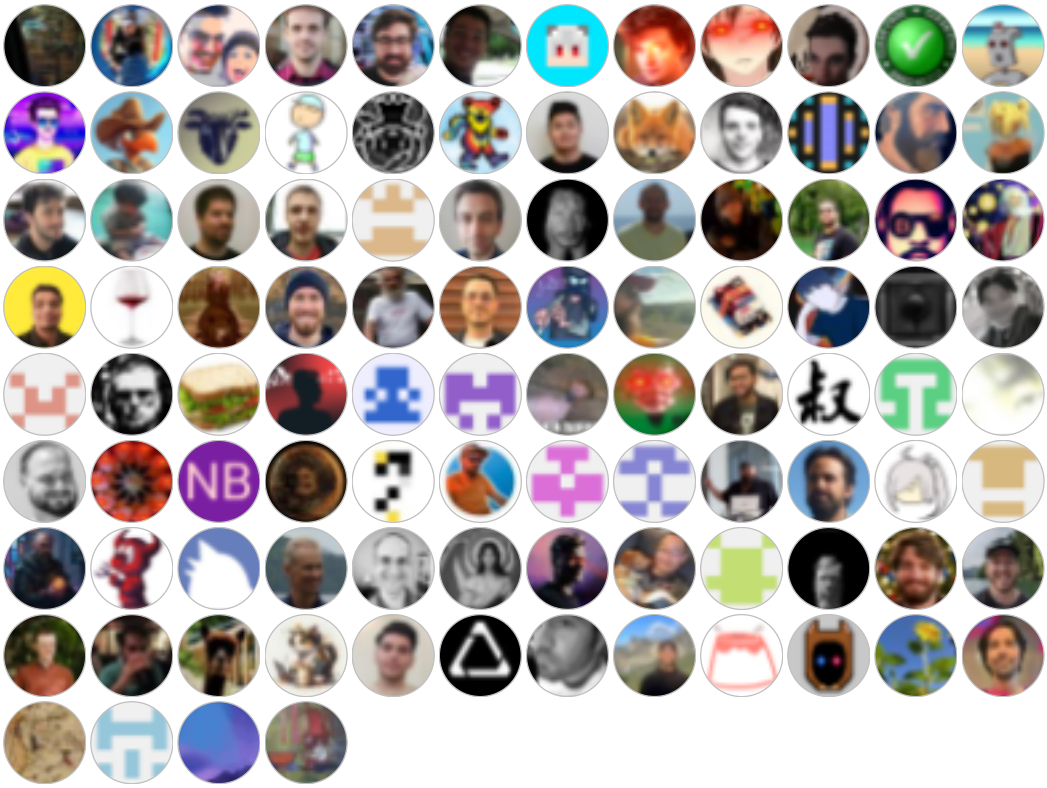
### Breaking Changes

Breaking Changes

### License

All NIPs are public domain.

### Contributors




## NIP-01

## Basic protocol flow description

`draft` `mandatory`

This NIP defines the basic protocol that should be implemented by everybody. New NIPs may add new optional (or mandatory) fields and messages and features to the structures and flows described here.

### Events and signatures

Each user has a keypair. Signatures, public key, and encodings are done according to the Schnorr signatures standard for the curve `secp256k1` .

The only object type that exists is the `event`, which has the following format on the wire:

```
{
  "id": <32-bytes lowercase hex-encoded sha256 of the serialized event data>,
  "pubkey": <32-bytes lowercase hex-encoded public key of the event creator>,
  "created_at": <unix timestamp in seconds>,
  "kind": <integer between 0 and 65535>,
  "tags": [
    [<arbitrary string>...],
    // ...
  ],
  "content": <arbitrary string>,
  "sig": <64-bytes lowercase hex of the signature of the sha256 hash of the
serialized event data, which is the same as the "id" field>
}
```

To obtain the `event.id`, we `sha256` the serialized event. The serialization is done over the UTF-8 JSON-serialized string (which is described below) of the following structure:

```
[
  0,
  <pubkey, as a lowercase hex string>,
  <created_at, as a number>,
  <kind, as a number>,
  <tags, as an array of arrays of non-null strings>,
  <content, as a string>
]
```

To prevent implementation differences from creating a different event ID for the same event, the following rules MUST be followed while serializing:

- UTF-8 should be used for encoding.
- Whitespace, line breaks or other unnecessary formatting should not be included in the output JSON.
- The following characters in the content field must be escaped as shown, and all other characters must be included verbatim:
  - A line break (`0x0A`), use `\n`
  - A double quote (`0x22`), use `\"`
  - A backslash (`0x5C`), use `\\`
  - A carriage return (`0x0D`), use `\r`
  - A tab character (`0x09`), use `\t`
  - A backspace, (`0x08`), use `\b`
  - A form feed, (`0x0C`), use `\f`

#### Tags

Each tag is an array of one or more strings, with some conventions around them. Take a look at the example below:

```
{
  "tags": [
    ["e", "5c83da77af1dec6d7289834998ad7aafbd9e2191396d75ec3cc27f5a77226f36",
"wss://nostr.example.com"],
    ["p", "f7234bd4c1394dda46d09f35bd384dd30cc552ad5541990f98844fb06676e9ca"],
    ["a",
"30023:f7234bd4c1394dda46d09f35bd384dd30cc552ad5541990f98844fb06676e9ca:abcd",
"wss://nostr.example.com"],
    ["alt", "reply"],
    // ...
  ],
  // ...
}
```

The first element of the tag array is referred to as the tag *name* or *key* and the second as the tag *value*. So we can safely say that the event above has an `e` tag set to `"5c83da77af1dec6d7289834998ad7aafbd9e2191396d75ec3cc27f5a77226f36"`, an `alt` tag set to `"reply"` and so on. All elements after the second do not have a conventional name.

This NIP defines 3 standard tags that can be used across all event kinds with the same meaning. They are as follows:

- The `e` tag, used to refer to an event: `["e", <32-bytes lowercase hex of the id of another event>, <recommended relay URL, optional>]`
- The `p` tag, used to refer to another user: `["p", <32-bytes lowercase hex of a pubkey>, <recommended relay URL, optional>]`
- The `a` tag, used to refer to an addressable or replaceable event
  - for an addressable event: `["a", <kind integer>:<32-bytes lowercase hex of a pubkey>:<d tag value>, <recommended relay URL, optional>]`
  - for a normal replaceable event: `["a", <kind integer>:<32-bytes lowercase hex of a pubkey>:, <recommended relay URL, optional>]`

As a convention, all single-letter (only english alphabet letters: a-z, A-Z) key tags are expected to be indexed by relays, such that it is possible, for example, to query or subscribe to events that reference the event `"5c83da77af1dec6d7289834998ad7aafbd9e2191396d75ec3cc27f5a77226f36"` by using the `{"#e":["5c83da77af1dec6d7289834998ad7aafbd9e2191396d75ec3cc27f5a77226f36"]}` filter. Only the first value in any given tag is indexed.

#### Kinds

Kinds specify how clients should interpret the meaning of each event and the other fields of each event (e.g. an `"r"` tag may have a meaning in an event of kind 1 and an entirely different meaning in an event of kind 10002). Each NIP may define the meaning of a set of kinds that weren't defined elsewhere. This NIP defines two basic kinds:

- `0`: **user metadata**: the `content` is set to a stringified JSON object `{name: <username>, about: <string>, picture: <url, string>}` describing the user who created the event. Extra metadata fields may be set. A relay may delete older events once it gets a new one for the same pubkey.
- `1`: **text note**: the `content` is set to the **plaintext** content of a note (anything the user wants to say). Content that must be parsed, such as Markdown and HTML, should not be used. Clients should also not parse content as those.

And also a convention for kind ranges that allow for easier experimentation and flexibility of relay implementation:

- for kind `n` such that `1000 <= n < 10000 || 4 <= n < 45 || n == 1 || n == 2`, events are **regular**, which means they're all expected to be stored by relays.
- for kind `n` such that `10000 <= n < 20000 || n == 0 || n == 3`, events are **replaceable**, which means that, for each combination of `pubkey` and `kind`, only the latest event MUST be stored by relays, older versions MAY be discarded.
- for kind `n` such that `20000 <= n < 30000`, events are **ephemeral**, which means they are not expected to be stored by relays.
- for kind `n` such that `30000 <= n < 40000`, events are **addressable** by their `kind`, `pubkey` and `d` tag value – which means that, for each combination of `kind`, `pubkey` and the `d` tag value, only the latest event MUST be stored by relays, older versions MAY be discarded.

In case of replaceable events with the same timestamp, the event with the lowest id (first in lexical order) should be retained, and the other discarded.

When answering to `REQ` messages for replaceable events such as `{"kinds":[0],"authors":[<hex-key>]}`, even if the relay has more than one version stored, it SHOULD return just the latest one.

These are just conventions and relay implementations may differ.

### Communication between clients and relays

Relays expose a websocket endpoint to which clients can connect. Clients SHOULD open a single websocket connection to each relay and use it for all their subscriptions. Relays MAY limit number of connections from specific IP/client/etc.

#### From client to relay: sending events and creating subscriptions

Clients can send 3 types of messages, which must be JSON arrays, according to the following patterns:

- `["EVENT", <event JSON as defined above>]`, used to publish events.

- `["REQ", <subscription_id>, <filters1>, <filters2>, ...]`, used to request events and subscribe to new updates.
- `["CLOSE", <subscription_id>]`, used to stop previous subscriptions.

`<subscription_id>` is an arbitrary, non-empty string of max length 64 chars. It represents a subscription per connection. Relays MUST manage `<subscription_id>`s independently for each WebSocket connection. `<subscription_id>`s are not guaranteed to be globally unique.

`<filtersX>` is a JSON object that determines what events will be sent in that subscription, it can have the following attributes:

```
{
  "ids": <a list of event ids>,
  "authors": <a list of lowercase pubkeys, the pubkey of an event must be one of
these>,
  "kinds": <a list of a kind numbers>,
  "#<single-letter (a-zA-Z)>": <a list of tag values, for #e — a list of event ids,
for #p — a list of pubkeys, etc.>,
  "since": <an integer unix timestamp in seconds. Events must have a created_at >=
to this to pass>,
  "until": <an integer unix timestamp in seconds. Events must have a created_at <=
to this to pass>,
  "limit": <maximum number of events relays SHOULD return in the initial query>
}
```

Upon receiving a `REQ` message, the relay SHOULD return events that match the filter. Any new events it receives SHOULD be sent to that same websocket until the connection is closed, a `CLOSE` event is received with the same `<subscription_id>`, or a new `REQ` is sent using the same `<subscription_id>` (in which case a new subscription is created, replacing the old one).

Filter attributes containing lists (`ids`, `authors`, `kinds` and tag filters like `#e`) are JSON arrays with one or more values. At least one of the arrays' values must match the relevant field in an event for the condition to be considered a match. For scalar event attributes such as `authors` and `kind`, the attribute from the event must be contained in the filter list. In the case of tag attributes such as `#e`, for which an event may have multiple values, the event and filter condition values must have at least one item in common.

The `ids`, `authors`, `#e` and `#p` filter lists MUST contain exact 64-character lowercase hex values.

The `since` and `until` properties can be used to specify the time range of events returned in the subscription. If a filter includes the `since` property, events with `created_at` greater than or equal to `since` are considered to match the filter. The `until` property is similar except that `created_at` must be less than or equal to `until`. In short, an event matches a filter if `since <= created_at <= until` holds.

All conditions of a filter that are specified must match for an event for it to pass the filter, i.e., multiple conditions are interpreted as `&&` conditions.

A `REQ` message may contain multiple filters. In this case, events that match any of the filters are to be returned, i.e., multiple filters are to be interpreted as `||` conditions.

The `limit` property of a filter is only valid for the initial query and MUST be ignored afterwards. When `limit: n` is present it is assumed that the events returned in the initial query will be the last `n` events ordered by the `created_at`. Newer events should appear first, and in the case of ties the event with the lowest id (first in lexical order) should be first. It is safe to return less events than `limit` specifies, but it is expected that relays do not return (much) more events than requested so clients don't get unnecessarily overwhelmed by data.

**From relay to client: sending events and notices**

Relays can send 5 types of messages, which must also be JSON arrays, according to the following patterns:

- `["EVENT", <subscription_id>, <event JSON as defined above>]`, used to send events requested by clients.
- `["OK", <event_id>, <true|false>, <message>]`, used to indicate acceptance or denial of an `EVENT` message.
- `["EOSE", <subscription_id>]`, used to indicate the *end of stored events* and the beginning of events newly received in real-time.
- `["CLOSED", <subscription_id>, <message>]`, used to indicate that a subscription was ended on the server side.
- `["NOTICE", <message>]`, used to send human-readable error messages or other things to clients.

This NIP defines no rules for how `NOTICE` messages should be sent or treated.

- `EVENT` messages MUST be sent only with a subscription ID related to a subscription previously initiated by the client (using the `REQ` message above).
- `OK` messages MUST be sent in response to `EVENT` messages received from clients, they must have the 3rd parameter set to `true` when an event has been accepted by the relay, `false` otherwise. The 4th parameter MUST always be present, but MAY be an empty string when the 3rd is `true`, otherwise it MUST be a string formed by a machine-readable single-word prefix followed by a `:` and then a human-readable message. Some examples:
  - `["OK", "b1a649ebe8...", true, ""]`
  - `["OK", "b1a649ebe8...", true, "pow: difficulty 25>=24"]`
  - `["OK", "b1a649ebe8...", true, "duplicate: already have this event"]`
  - `["OK", "b1a649ebe8...", false, "blocked: you are banned from posting here"]`
  - `["OK", "b1a649ebe8...", false, "blocked: please register your pubkey at https://my-expensive-relay.example.com"]`
  - `["OK", "b1a649ebe8...", false, "rate-limited: slow down there chief"]`
  - `["OK", "b1a649ebe8...", false, "invalid: event creation date is too far off from the current time"]`
  - `["OK", "b1a649ebe8...", false, "pow: difficulty 26 is less than 30"]`
  - `["OK", "b1a649ebe8...", false, "error: could not connect to the database"]`
- `CLOSED` messages MUST be sent in response to a `REQ` when the relay refuses to fulfill it. It can also be sent when a relay decides to kill a subscription on its side before a client has disconnected or sent a `CLOSE`. This message uses the same pattern of `OK` messages with the machine-readable prefix and human-readable message. Some examples:
  - `["CLOSED", "sub1", "unsupported: filter contains unknown elements"]`
  - `["CLOSED", "sub1", "error: could not connect to the database"]`
  - `["CLOSED", "sub1", "error: shutting down idle subscription"]`
- The standardized machine-readable prefixes for `OK` and `CLOSED` are: `duplicate`, `pow`, `blocked`, `rate-limited`, `invalid`, and `error` for when none of that fits.

## NIP-02

## Follow List

`final` `optional`

A special event with kind `3`, meaning “follow list” is defined as having a list of `p` tags, one for each of the followed/known profiles one is following.

Each tag entry should contain the key for the profile, a relay URL where events from that key can be found (can be set to an empty string if not needed), and a local name (or “petname”) for that profile (can also be set to an empty string or not provided), i.e., `["p", <32-bytes hex key>, <main relay URL>, <petname>]`.

The `.content` is not used.

For example:

```
{
  "kind": 3,
  "tags": [
    ["p", "91cf9..4e5ca", "wss://alicerelay.com/", "alice"],
    ["p", "14aeb..8dad4", "wss://bobrelay.com/nostr", "bob"],
    ["p", "612ae..e610f", "ws://carolrelay.com/ws", "carol"]
  ],
  "content": "",
  // other fields...
}
```

Every new following list that gets published overwrites the past ones, so it should contain all entries. Relays and clients SHOULD delete past following lists as soon as they receive a new one.

Whenever new follows are added to an existing list, clients SHOULD append them to the end of the list, so they are stored in chronological order.

### Uses

#### Follow list backup

If one believes a relay will store their events for sufficient time, they can use this kind-3 event to backup their following list and recover on a different device.

**Profile discovery and context augmentation**

A client may rely on the kind-3 event to display a list of followed people by profiles one is browsing; make lists of suggestions on who to follow based on the follow lists of other people one might be following or browsing; or show the data in other contexts.

**Relay sharing**

A client may publish a follow list with good relays for each of their follows so other clients may use these to update their internal relay lists if needed, increasing censorship-resistance.

**Petname scheme**

The data from these follow lists can be used by clients to construct local "petname" tables derived from other people's follow lists. This alleviates the need for global human-readable names. For example:

A user has an internal follow list that says

```
[
  ["p", "21df6d143fb96c2ec9d63726bf9edc71", "", "erin"]
]
```

And receives two follow lists, one from `21df6d143fb96c2ec9d63726bf9edc71` that says

```
[
  ["p", "a8bb3d884d5d90b413d9891fe4c4e46d", "", "david"]
]
```

and another from `a8bb3d884d5d90b413d9891fe4c4e46d` that says

```
[
  ["p", "f57f54057d2a7af0efecc8b0b66f5708", "", "frank"]
]
```

When the user sees `21df6d143fb96c2ec9d63726bf9edc71` the client can show *erin* instead; When the user sees `a8bb3d884d5d90b413d9891fe4c4e46d` the client can show *david.erin* instead; When the user sees `f57f54057d2a7af0efecc8b0b66f5708` the client can show *frank.david.erin* instead.

## NIP-03

## OpenTimestamps Attestations for Events

`draft` `optional`

This NIP defines an event with `kind:1040` that can contain an OpenTimestamps proof for any other event:

```
{
  "kind": 1040
  "tags": [
    ["e", <event-id>, <relay-url>],
    ["alt", "opentimestamps attestation"]
  ],
  "content": <base64-encoded OTS file data>
}
```

- The OpenTimestamps proof MUST prove the referenced `e` event id as its digest.
- The `content` MUST be the full content of an `.ots` file containing at least one Bitcoin attestation. This file SHOULD contain a **single** Bitcoin attestation (as not more than one valid attestation is necessary and less bytes is better than more) and no reference to "pending" attestations since they are useless in this context.

**Example OpenTimestamps proof verification flow**

Using `nak`, `jq` and `ots`:

```
~> nak req -i e71c6ea722987debdb60f81f9ea4f604b5ac0664120dd64fb9d23abc4ec7c323
wss://nostr-pub.wellorder.net | jq -r .content | ots verify
```

```
> using an esplora server at https://blockstream.info/api
- sequence ending on block 810391 is valid
timestamp validated at block [810391]
```

**Warning** `unrecommended`: deprecated in favor of NIP-17

## NIP-04

## Encrypted Direct Message

`final` `unrecommended` `optional`

A special event with kind `4`, meaning "encrypted direct message". It is supposed to have the following attributes:

**`content`** MUST be equal to the base64-encoded, aes-256-cbc encrypted string of anything a user wants to write, encrypted using a shared cipher generated by combining the recipient's public-key with the sender's private-key; this appended by the base64-encoded initialization vector as if it was a querystring parameter named "iv". The format is the following: `"content": "<encrypted_text>?iv=<initialization_vector>"`.

**`tags`** MUST contain an entry identifying the receiver of the message (such that relays may naturally forward this event to them), in the form `["p", "<pubkey, as a hex string>"]`.

**`tags`** MAY contain an entry identifying the previous message in a conversation or a message we are explicitly replying to (such that contextual, more organized conversations may happen), in the form `["e", "<event_id>"]`.

**Note**: By default in the libsecp256k1 ECDH implementation, the secret is the SHA256 hash of the shared point (both X and Y coordinates). In Nostr, only the X coordinate of the shared point is used as the secret and it is NOT hashed. If using libsecp256k1, a custom function that copies the X coordinate must be passed as the `hashfp` argument in `secp256k1_ecdh`. See here .

Code sample for generating such an event in JavaScript:

```
import crypto from 'crypto'
import * as secp from '@noble/secp256k1'

let sharedPoint = secp.getSharedSecret(ourPrivateKey, '02' + theirPublicKey)
let sharedX = sharedPoint.slice(1, 33)

let iv = crypto.randomFillSync(new Uint8Array(16))
var cipher = crypto.createCipheriv(
  'aes-256-cbc',
  Buffer.from(sharedX),
  iv
)
let encryptedMessage = cipher.update(text, 'utf8', 'base64')
encryptedMessage += cipher.final('base64')
let ivBase64 = Buffer.from(iv.buffer).toString('base64')

let event = {
  pubkey: ourPubKey,
  created_at: Math.floor(Date.now() / 1000),
  kind: 4,
  tags: [['p', theirPublicKey]],
  content: encryptedMessage + '?iv=' + ivBase64
}
```

### Security Warning

This standard does not go anywhere near what is considered the state-of-the-art in encrypted communication between peers, and it leaks metadata in the events, therefore it must not be used for anything you really need to keep secret, and only with relays that use `AUTH` to restrict who can fetch your `kind:4` events.

### Client Implementation Warning

Clients *should not* search and replace public key or note references from the `.content`. If processed like a regular text note (where `@npub...` is replaced with `#[0]` with a `["p", "..."]` tag) the tags are leaked and the mentioned user will receive the message in their inbox.

## NIP-05

## Mapping Nostr keys to DNS-based internet identifiers

`final` `optional`

On events of kind `0` (`user metadata`) one can specify the key `"nip05"` with an internet identifier (an email-like address) as the value. Although there is a link to a very liberal "internet identifier" specification above, NIP-05 assumes the `<local-part>` part will be restricted to the characters `a-z0-9-_.`, case-insensitive.

Upon seeing that, the client splits the identifier into `<local-part>` and `<domain>` and use these values to make a GET request to `https://<domain>/.well-known/nostr.json?name=<local-part>`.

The result should be a JSON document object with a key `"names"` that should then be a mapping of names to hex formatted public keys. If the public key for the given `<name>` matches the `pubkey` from the `user metadata` event, the client then concludes that the given pubkey can indeed be referenced by its identifier.

**Example**

If a client sees an event like this:

```
{
  "pubkey": "b0635d6a9851d3aed0cd6c495b282167acf761729078d975fc341b22650b07b9",
  "kind": 0,
  "content": "{\"name\": \"bob\", \"nip05\": \"bob@example.com\"}"
  // other fields...
}
```

It will make a GET request to `https://example.com/.well-known/nostr.json?name=bob` and get back a response that will look like

```
{
  "names": {
    "bob": "b0635d6a9851d3aed0cd6c495b282167acf761729078d975fc341b22650b07b9"
  }
}
```

or with the **recommended** `"relays"` attribute:

```
{
  "names": {
    "bob": "b0635d6a9851d3aed0cd6c495b282167acf761729078d975fc341b22650b07b9"
  },
  "relays": {
    "b0635d6a9851d3aed0cd6c495b282167acf761729078d975fc341b22650b07b9": [
"wss://relay.example.com", "wss://relay2.example.com" ]
  }
}
```

If the pubkey matches the one given in `"names"` (as in the example above) that means the association is right and the `"nip05"` identifier is valid and can be displayed.

The recommended `"relays"` attribute may contain an object with public keys as properties and arrays of relay URLs as values. When present, that can be used to help clients learn in which relays the specific user may be found. Web servers which serve `/.well-known/nostr.json` files dynamically based on the query string SHOULD also serve the relays data for any name they serve in the same reply when that is available.

### Finding users from their NIP-05 identifier

A client may implement support for finding users' public keys from *internet identifiers*, the flow is the same as above, but reversed: first the client fetches the *well-known* URL and from there it gets the public key of the user, then it tries to fetch the kind `0` event for that user and check if it has a matching `"nip05"`.

### Notes

**Identification, not verification**

The NIP-05 is not intended to *verify* a user, but only to *identify* them, for the purpose of facilitating the exchange of a contact or their search.
Exceptions are people who own (e.g., a company) or are connected (e.g., a project) to a well-known domain, who can

exploit NIP-05 as an attestation of their relationship with it, and thus to the organization behind it, thereby gaining an element of trust.

**User discovery implementation suggestion**

A client can use this to allow users to search other profiles. If a client has a search box or something like that, a user may be able to type "bob@example.com " there and the client would recognize that and do the proper queries to obtain a pubkey and suggest that to the user.

**Clients must always follow public keys, not NIP-05 addresses**

For example, if after finding that `bob@bob.com` has the public key `abc...def`, the user clicks a button to follow that profile, the client must keep a primary reference to `abc...def`, not `bob@bob.com`. If, for any reason, the address `https://bob.com/.well-known/nostr.json?name=bob` starts returning the public key `1d2...e3f` at any time in the future, the client must not replace `abc...def` in his list of followed profiles for the user (but it should stop displaying "bob@bob.com " for that user, as that will have become an invalid `"nip05"` property).

**Public keys must be in hex format**

Keys must be returned in hex format. Keys in NIP-19 `npub` format are only meant to be used for display in client UIs, not in this NIP.

**Showing just the domain as an identifier**

Clients may treat the identifier `_@domain` as the "root" identifier, and choose to display it as just the `<domain>`. For example, if Bob owns `bob.com`, he may not want an identifier like `bob@bob.com` as that is redundant. Instead, Bob can use the identifier `_@bob.com` and expect Nostr clients to show and treat that as just `bob.com` for all purposes.

**Reasoning for the `/.well-known/nostr.json?name=<local-part>` format**

By adding the `<local-part>` as a query string instead of as part of the path, the protocol can support both dynamic servers that can generate JSON on-demand and static servers with a JSON file in it that may contain multiple names.

**Allowing access from JavaScript apps**

JavaScript Nostr apps may be restricted by browser CORS policies that prevent them from accessing `/.well-known/nostr.json` on the user's domain. When CORS prevents JS from loading a resource, the JS program sees it as a network failure identical to the resource not existing, so it is not possible for a pure-JS app to tell the user for certain that the failure was caused by a CORS issue. JS Nostr apps that see network failures requesting `/.well-known/nostr.json` files may want to recommend to users that they check the CORS policy of their servers, e.g.:

```
$ curl -sI https://example.com/.well-known/nostr.json?name=bob | grep -i ^Access-Control
Access-Control-Allow-Origin: *
```

Users should ensure that their `/.well-known/nostr.json` is served with the HTTP header `Access-Control-Allow-Origin: *` to ensure it can be validated by pure JS apps running in modern browsers.

**Security Constraints**

The `/.well-known/nostr.json` endpoint MUST NOT return any HTTP redirects.

Fetchers MUST ignore any HTTP redirects given by the `/.well-known/nostr.json` endpoint.

# NIP-06

# Basic key derivation from mnemonic seed phrase

`draft` `optional`

BIP39 is used to generate mnemonic seed words and derive a binary seed from them.

BIP32 is used to derive the path `m/44'/1237'/<account>'/0/0` (according to the Nostr entry on SLIP44 ).

A basic client can simply use an `account` of `0` to derive a single key. For more advanced use-cases you can increment `account`, allowing generation of practically infinite keys from the 5-level path with hardened derivation.

Other types of clients can still get fancy and use other derivation paths for their own other purposes.

#### Test vectors

mnemonic: leader monkey parrot ring guide accident before fence cannon height naive bean
private key (hex): 7f7ff03d123792d6ac594bfa67bf6d0c0ab55b6b1fdb6249303fe861f1ccba9a
nsec: nsec10allq0gjx7fddtzef0ax00mdps9t2kmtrldkyjfs8l5xruwvh2dq0lhhkp
public key (hex): 17162c921dc4d2518f9a101db33695df1afb56ab82f5ff3e5da6eec3ca5cd917
npub: npub1zutzeysacnf9rru6zqwmxd54mud0k44tst6l70ja5mhv8jjumytsd2x7nu

---

mnemonic: what bleak badge arrange retreat wolf trade produce cricket blur garlic valid proud rude strong choose busy staff weather area salt hollow arm fade
private key (hex): c15d739894c81a2fcfd3a2df85a0d2c0dbc47a280d092799f144d73d7ae78add
nsec: nsec1c9wh8xy5eqdzln7n5t0ctgxjcrdug73gp5yj0x03gntn67h83twssdfhel
public key (hex): d41b22899549e1f3d335a31002cfd382174006e166d3e658e3a5eecdb6463573
npub: npub16sdj9zv4f8sl85e45vgq9n7nsgt5qphpvmf7vk8r5hhvmdjxx4es8rq74h

## NIP-07

## `window.nostr` capability for web browsers

`draft` `optional`

The `window.nostr` object may be made available by web browsers or extensions and websites or web-apps may make use of it after checking its availability.

That object must define the following methods:

```
async window.nostr.getPublicKey(): string // returns a public key as hex
async window.nostr.signEvent(event: { created_at: number, kind: number, tags: string[][], content: string }): Event // takes an event object, adds `id`, `pubkey` and `sig` and returns it
```

Aside from these two basic above, the following functions can also be implemented optionally:

```
async window.nostr.getRelays(): { [url: string]: {read: boolean, write: boolean} }
// returns a basic map of relay urls to relay policies
async window.nostr.nip04.encrypt(pubkey, plaintext): string // returns ciphertext and iv as specified in nip-04 (deprecated)
async window.nostr.nip04.decrypt(pubkey, ciphertext): string // takes ciphertext and iv as specified in nip-04 (deprecated)
async window.nostr.nip44.encrypt(pubkey, plaintext): string // returns ciphertext as specified in nip-44
async window.nostr.nip44.decrypt(pubkey, ciphertext): string // takes ciphertext as specified in nip-44
```

#### Recommendation to Extension Authors

To make sure that the `window.nostr` is available to nostr clients on page load, the authors who create Chromium and Firefox extensions should load their scripts by specifying `"run_at": "document_end"` in the extension's manifest.

#### Implementation

See https://github.com/aljazceru/awesome-nostr#nip-07-browser-extensions .

> **Warning** `unrecommended`: deprecated in favor of NIP-27

## NIP-08

## Handling Mentions

`final` `unrecommended` `optional`

This document standardizes the treatment given by clients of inline mentions of other events and pubkeys inside the content of `text_note`s.

Clients that want to allow tagged mentions they MUST show an autocomplete component or something analogous to that whenever the user starts typing a special key (for example, "@") or presses some button to include a mention etc – or these clients can come up with other ways to unambiguously differentiate between mentions and normal text.

Once a mention is identified, for example, the pubkey `27866e9d854c78ae625b867eefdfa9580434bc3e675be08d2acb526610d96fbe`, the client MUST add that pubkey to the `.tags` with the tag `p`, then replace its textual reference (inside `.content`) with the notation `#[index]` in which "index" is equal to the 0-based index of the related tag in the tags array.

The same process applies for mentioning event IDs.

A client that receives a `text_note` event with such `#[index]` mentions in its `.content` CAN do a search-and-replace using the actual contents from the `.tags` array with the actual pubkey or event ID that is mentioned, doing any desired context augmentation (for example, linking to the pubkey or showing a preview of the mentioned event contents) it wants in the process.

Where `#[index]` has an `index` that is outside the range of the tags array or points to a tag that is not an `e` or `p` tag or a tag otherwise declared to support this notation, the client MUST NOT perform such replacement or augmentation, but instead display it as normal text.

## NIP-09

## Event Deletion Request

`draft` `optional`

A special event with kind `5`, meaning "deletion request" is defined as having a list of one or more `e` or `a` tags, each referencing an event the author is requesting to be deleted. Deletion requests SHOULD include a `k` tag for the kind of each event being requested for deletion.

The event's `content` field MAY contain a text note describing the reason for the deletion request.

For example:

```
{
  "kind": 5,
  "pubkey": <32-bytes hex-encoded public key of the event creator>,
  "tags": [
    ["e", "dcd59..464a2"],
    ["e", "968c5..ad7a4"],
    ["a", "<kind>:<pubkey>:<d-identifier>"],
    ["k", "1"],
    ["k", "30023"]
  ],
  "content": "these posts were published by accident",
  // other fields...
}
```

Relays SHOULD delete or stop publishing any referenced events that have an identical `pubkey` as the deletion request. Clients SHOULD hide or otherwise indicate a deletion request status for referenced events.

Relays SHOULD continue to publish/share the deletion request events indefinitely, as clients may already have the event that's intended to be deleted. Additionally, clients SHOULD broadcast deletion request events to other relays which don't have it.

When an `a` tag is used, relays SHOULD delete all versions of the replaceable event up to the `created_at` timestamp of the deletion request event.

### Client Usage

Clients MAY choose to fully hide any events that are referenced by valid deletion request events. This includes text notes, direct messages, or other yet-to-be defined event kinds. Alternatively, they MAY show the event along with an icon or other indication that the author has "disowned" the event. The `content` field MAY also be used to replace the deleted events' own content, although a user interface should clearly indicate that this is a deletion request reason, not the original content.

A client MUST validate that each event `pubkey` referenced in the `e` tag of the deletion request is identical to the deletion request `pubkey`, before hiding or deleting any event. Relays can not, in general, perform this validation and should not be treated as authoritative.

Clients display the deletion request event itself in any way they choose, e.g., not at all, or with a prominent notice.

Clients MAY choose to inform the user that their request for deletion does not guarantee deletion because it is impossible to delete events from all relays and clients.

### Relay Usage

Relays MAY validate that a deletion request event only references events that have the same `pubkey` as the deletion request itself, however this is not required since relays may not have knowledge of all referenced events.

### Deletion Request of a Deletion Request

Publishing a deletion request event against a deletion request has no effect. Clients and relays are not obliged to support “unrequest deletion” functionality.

## NIP-10

## On “e” and “p” tags in Text Events (kind 1)

`draft` `optional`

### Abstract

This NIP describes how to use “e” and “p” tags in text events, especially those that are replies to other text events. It helps clients thread the replies into a tree rooted at the original event.

### Marked “e” tags (PREFERRED)

`["e", <event-id>, <relay-url>, <marker>, <pubkey>]`

Where:

- `<event-id>` is the id of the event being referenced.
- `<relay-url>` is the URL of a recommended relay associated with the reference. Clients SHOULD add a valid `<relay-url>` field, but may instead leave it as `""`.
- `<marker>` is optional and if present is one of `"reply"`, `"root"`, or `"mention"`.
- `<pubkey>` is optional, SHOULD be the pubkey of the author of the referenced event

Those marked with `"reply"` denote the id of the reply event being responded to. Those marked with `"root"` denote the root id of the reply thread being responded to. For top level replies (those replying directly to the root event), only the `"root"` marker should be used. Those marked with `"mention"` denote a quoted or reposted event id.

A direct reply to the root of a thread should have a single marked “e” tag of type “root”.

> This scheme is preferred because it allows events to mention others without confusing them with `<reply-id>` or `<root-id>`.

`<pubkey>` SHOULD be the pubkey of the author of the `e` tagged event, this is used in the outbox model to search for that event from the authors write relays where relay hints did not resolve the event.

### The “p” tag

Used in a text event contains a list of pubkeys used to record who is involved in a reply thread.

When replying to a text event E the reply event’s “p” tags should contain all of E’s “p” tags as well as the `"pubkey"` of the event being replied to.

Example: Given a text event authored by `a1` with “p” tags [`p1`, `p2`, `p3`] then the “p” tags of the reply should be [`a1`, `p1`, `p2`, `p3`] in no particular order.

### Deprecated Positional “e” tags

This scheme is not in common use anymore and is here just to keep backward compatibility with older events on the network.

Positional `e` tags are deprecated because they create ambiguities that are difficult, or impossible to resolve when an event references another but is not a reply.

They use simple `e` tags without any marker.

`["e", <event-id>, <relay-url>]` as per NIP-01.

Where:

- `<event-id>` is the id of the event being referenced.
- `<relay-url>` is the URL of a recommended relay associated with the reference. Many clients treat this field as optional.

**The positions of the “e” tags within the event denote specific meanings as follows**:

- No “e” tag:
  This event is not a reply to, nor does it refer to, any other event.
- One “e” tag:
  `["e", <id>]`: The id of the event to which this event is a reply.
- Two “e” tags: `["e", <root-id>]`, `["e", <reply-id>]`
  `<root-id>` is the id of the event at the root of the reply chain. `<reply-id>` is the id of the article to which this event is a reply.
- Many “e” tags: `["e", <root-id>]` `["e", <mention-id>]`, ..., `["e", <reply-id>]`
  There may be any number of `<mention-ids>`. These are the ids of events which may, or may not be in the reply chain. They are citing from this event. `root-id` and `reply-id` are as above. NIP-11 ======

## Relay Information Document

`draft` `optional`

Relays may provide server metadata to clients to inform them of capabilities, administrative contacts, and various server attributes. This is made available as a JSON document over HTTP, on the same URI as the relay’s websocket.

When a relay receives an HTTP(s) request with an `Accept` header of `application/nostr+json` to a URI supporting WebSocket upgrades, they SHOULD return a document with the following structure.

```
{
  "name": <string identifying relay>,
  "description": <string with detailed information>,
  "banner": <a link to an image (e.g. in .jpg, or .png format)>,
  "icon": <a link to an icon (e.g. in .jpg, or .png format>,
  "pubkey": <administrative contact pubkey>,
  "contact": <administrative alternate contact>,
  "supported_nips": <a list of NIP numbers supported by the relay>,
  "software": <string identifying relay software URL>,
  "version": <string version identifier>
}
```

Any field may be omitted, and clients MUST ignore any additional fields they do not understand. Relays MUST accept CORS requests by sending `Access-Control-Allow-Origin`, `Access-Control-Allow-Headers`, and `Access-Control-Allow-Methods` headers.

## Field Descriptions

**Name**

A relay may select a `name` for use in client software. This is a string, and SHOULD be less than 30 characters to avoid client truncation.

**Description**

Detailed plain-text information about the relay may be contained in the `description` string. It is recommended that this contain no markup, formatting or line breaks for word wrapping, and simply use double newline characters to separate paragraphs. There are no limitations on length.

**Banner**

To make nostr relay management more user friendly, an effort should be made by relay owners to communicate with non-dev non-technical nostr end users. A banner is a visual representation of the relay. It should aim to visually communicate the brand of the relay, complementing the text `Description`. Here is an example banner mockup as visualized in Damus iOS relay view of the Damus relay.

**Icon**

Icon is a compact visual representation of the relay for use in UI with limited real estate such as a nostr user’s relay list view. Below is an example URL pointing to an image to be used as an icon for the relay. Recommended to be squared in shape.

```
{
  "icon":
"https://nostr.build/i/53866b44135a27d624e99c6165cabd76ac8f72797209700acb189fce75021f47.jpg",

  // other fields...
}
```

#### Pubkey

An administrative contact may be listed with a `pubkey`, in the same format as Nostr events (32-byte hex for a `secp256k1` public key). If a contact is listed, this provides clients with a recommended address to send encrypted direct messages (See NIP-17 ) to a system administrator. Expected uses of this address are to report abuse or illegal content, file bug reports, or request other technical assistance.

Relay operators have no obligation to respond to direct messages.

#### Contact

An alternative contact may be listed under the `contact` field as well, with the same purpose as `pubkey`. Use of a Nostr public key and direct message SHOULD be preferred over this. Contents of this field SHOULD be a URI, using schemes such as `mailto` or `https` to provide users with a means of contact.

#### Supported NIPs

As the Nostr protocol evolves, some functionality may only be available by relays that implement a specific `NIP`. This field is an array of the integer identifiers of `NIP`s that are implemented in the relay. Examples would include `1`, for `"NIP-01"` and `9`, for `"NIP-09"`. Client-side `NIPs` SHOULD NOT be advertised, and can be ignored by clients.

#### Software

The relay server implementation MAY be provided in the `software` attribute. If present, this MUST be a URL to the project's homepage.

#### Version

The relay MAY choose to publish its software version as a string attribute. The string format is defined by the relay implementation. It is recommended this be a version number or commit identifier.

## Extra Fields

#### Server Limitations

These are limitations imposed by the relay on clients. Your client should expect that requests which exceed these *practical* limitations are rejected or fail immediately.

```
{
  "limitation": {
    "max_message_length": 16384,
    "max_subscriptions": 20,
    "max_filters": 100,
    "max_limit": 5000,
    "max_subid_length": 100,
    "max_event_tags": 100,
    "max_content_length": 8196,
    "min_pow_difficulty": 30,
    "auth_required": true,
    "payment_required": true,
    "restricted_writes": true,
    "created_at_lower_limit": 31536000,
    "created_at_upper_limit": 3
  },
  // other fields...
}
```

- `max_message_length`: this is the maximum number of bytes for incoming JSON that the relay will attempt to decode and act upon. When you send large subscriptions, you will be limited by this value. It also effectively

limits the maximum size of any event. Value is calculated from `[` to `]` and is after UTF-8 serialization (so some unicode characters will cost 2-3 bytes). It is equal to the maximum size of the WebSocket message frame.

- `max_subscriptions`: total number of subscriptions that may be active on a single websocket connection to this relay. It's possible that authenticated clients with a (paid) relationship to the relay may have higher limits.
- `max_filters`: maximum number of filter values in each subscription. Must be one or higher.
- `max_subid_length`: maximum length of subscription id as a string.
- `max_limit`: the relay server will clamp each filter's `limit` value to this number. This means the client won't be able to get more than this number of events from a single subscription filter. This clamping is typically done silently by the relay, but with this number, you can know that there are additional results if you narrowed your filter's time range or other parameters.
- `max_event_tags`: in any event, this is the maximum number of elements in the `tags` list.
- `max_content_length`: maximum number of characters in the `content` field of any event. This is a count of unicode characters. After serializing into JSON it may be larger (in bytes), and is still subject to the `max_message_length`, if defined.
- `min_pow_difficulty`: new events will require at least this difficulty of PoW, based on NIP-13 , or they will be rejected by this server.
- `auth_required`: this relay requires NIP-42 authentication to happen before a new connection may perform any other action. Even if set to False, authentication may be required for specific actions.
- `payment_required`: this relay requires payment before a new connection may perform any action.
- `restricted_writes`: this relay requires some kind of condition to be fulfilled in order to accept events (not necessarily, but including `payment_required` and `min_pow_difficulty`). This should only be set to `true` when users are expected to know the relay policy before trying to write to it – like belonging to a special pubkey-based whitelist or writing only events of a specific niche kind or content. Normal anti-spam heuristics, for example, do not qualify.
- `created_at_lower_limit`: 'created_at' lower limit
- `created_at_upper_limit`: 'created_at' upper limit

**Event Retention**

There may be a cost associated with storing data forever, so relays may wish to state retention times. The values stated here are defaults for unauthenticated users and visitors. Paid users would likely have other policies.

Retention times are given in seconds, with `null` indicating infinity. If zero is provided, this means the event will not be stored at all, and preferably an error will be provided when those are received.

```
{
  "retention": [
    {"kinds": [0, 1, [5, 7], [40, 49]], "time": 3600},
    {"kinds": [[40000, 49999]], "time": 100},
    {"kinds": [[30000, 39999]], "count": 1000},
    {"time": 3600, "count": 10000}
  ],
  // other fields...
}
```

`retention` is a list of specifications: each will apply to either all kinds, or a subset of kinds. Ranges may be specified for the kind field as a tuple of inclusive start and end values. Events of indicated kind (or all) are then limited to a `count` and/or time period.

It is possible to effectively blacklist Nostr-based protocols that rely on a specific `kind` number, by giving a retention time of zero for those `kind` values. While that is unfortunate, it does allow clients to discover servers that will support their protocol quickly via a single HTTP fetch.

There is no need to specify retention times for *ephemeral events* since they are not retained.

**Content Limitations**

Some relays may be governed by the arbitrary laws of a nation state. This may limit what content can be stored in clear-text on those relays. All clients are encouraged to use encryption to work around this limitation.

It is not possible to describe the limitations of each country's laws and policies which themselves are typically vague and constantly shifting.

Therefore, this field allows the relay operator to indicate which countries' laws might end up being enforced on them, and then indirectly on their users' content.

Users should be able to avoid relays in countries they don't like, and/or select relays in more favorable zones. Exposing this flexibility is up to the client software.

```
{
  "relay_countries": [ "CA", "US" ],
  // other fields...
}
```

- `relay_countries`: a list of two-level ISO country codes (ISO 3166-1 alpha-2) whose laws and policies may affect this relay. `EU` may be used for European Union countries.

Remember that a relay may be hosted in a country which is not the country of the legal entities who own the relay, so it's very likely a number of countries are involved.

**Community Preferences**

For public text notes at least, a relay may try to foster a local community. This would encourage users to follow the global feed on that relay, in addition to their usual individual follows. To support this goal, relays MAY specify some of the following values.

```
{
  "language_tags": ["en", "en-419"],
  "tags": ["sfw-only", "bitcoin-only", "anime"],
  "posting_policy": "https://example.com/posting-policy.html",
  // other fields...
}
```

- `language_tags` is an ordered list of IETF language tags indicating the major languages spoken on the relay.
- `tags` is a list of limitations on the topics to be discussed. For example `sfw-only` indicates that only "Safe For Work" content is encouraged on this relay. This relies on assumptions of what the "work" "community" feels "safe" talking about. In time, a common set of tags may emerge that allow users to find relays that suit their needs, and client software will be able to parse these tags easily. The `bitcoin-only` tag indicates that any *altcoin*, *"crypto"* or *blockchain* comments will be ridiculed without mercy.
- `posting_policy` is a link to a human-readable page which specifies the community policies for the relay. In cases where `sfw-only` is True, it's important to link to a page which gets into the specifics of your posting policy.

The `description` field should be used to describe your community goals and values, in brief. The `posting_policy` is for additional detail and legal terms. Use the `tags` field to signify limitations on content, or topics to be discussed, which could be machine processed by appropriate client software.

**Pay-to-Relay**

Relays that require payments may want to expose their fee schedules.

```
{
  "payments_url": "https://my-relay/payments",
  "fees": {
    "admission": [{ "amount": 1000000, "unit": "msats" }],
    "subscription": [{ "amount": 5000000, "unit": "msats", "period": 2592000 }],
    "publication": [{ "kinds": [4], "amount": 100, "unit": "msats" }],
  },
  // other fields...
}
```

**Examples**

As of 2 May 2023 the following command provided these results:

```
$ curl -H "Accept: application/nostr+json" https://eden.nostr.land | jq
```

```
{
  "description": "nostr.land family of relays (us-or-01)",
  "name": "nostr.land",
  "pubkey": "52b4a076bcbbbdc3a1aefa3735816cf74993b1b8db202b01c883c58be7fad8bd",
  "software": "custom",
  "supported_nips": [
    1,
    2,
    4,
    9,
    11,
    12,
    16,
    20,
    22,
    28,
    33,
    40
  ],
  "version": "1.0.1",
  "limitation": {
    "payment_required": true,
    "max_message_length": 65535,
    "max_event_tags": 2000,
    "max_subscriptions": 20,
    "auth_required": false
  },
  "payments_url": "https://eden.nostr.land",
  "fees": {
    "subscription": [
      {
        "amount": 2500000,
        "unit": "msats",
        "period": 2592000
      }
    ]
  },
}
```

## NIP-12

## Generic Tag Queries

`final` `mandatory`

Moved to NIP-01 .

## NIP-13

## Proof of Work

`draft` `optional`

This NIP defines a way to generate and interpret Proof of Work for nostr notes. Proof of Work (PoW) is a way to add a proof of computational work to a note. This is a bearer proof that all relays and clients can universally validate with a small amount of code. This proof can be used as a means of spam deterrence.

`difficulty` is defined to be the number of leading zero bits in the `NIP-01` id. For example, an id of `000000000e9d97a1ab09fc381030b346cdd7a142ad57e6df0b46dc9bef6c7e2d` has a difficulty of `36` with `36` leading 0 bits.

`002f...` is `0000 0000 0010 1111...` in binary, which has 10 leading zeroes. Do not forget to count leading zeroes for hex digits <= `7`.

## Mining

To generate PoW for a `NIP-01` note, a `nonce` tag is used:

```
{"content": "It's just me mining my own business", "tags": [["nonce", "1", "21"]]}
```

When mining, the second entry to the nonce tag is updated, and then the id is recalculated (see NIP-01 ). If the id has the desired number of leading zero bits, the note has been mined. It is recommended to update the `created_at` as well during this process.

The third entry to the nonce tag `SHOULD` contain the target difficulty. This allows clients to protect against situations where bulk spammers targeting a lower difficulty get lucky and match a higher difficulty. For example, if you require 40 bits to reply to your thread and see a committed target of 30, you can safely reject it even if the note has 40 bits difficulty. Without a committed target difficulty you could not reject it. Committing to a target difficulty is something all honest miners should be ok with, and clients `MAY` reject a note matching a target difficulty if it is missing a difficulty commitment.

## Example mined note

```
{
  "id": "000006d8c378af1779d2feebc7603a125d99eca0ccf1085959b307f64e5dd358",
  "pubkey": "a48380f4cfcc1ad5378294fcac36439770f9c878dd880ffa94bb74ea54a6f243",
  "created_at": 1651794653,
  "kind": 1,
  "tags": [
    ["nonce", "776797", "20"]
  ],
  "content": "It's just me mining my own business",
  "sig":
"284622fc0a3f4f1303455d5175f7ba962a3300d136085b9566801bc2e0699de0c7e31e44c81fb40ad9049173742e904713c35

}
```

## Validating

Here is some reference C code for calculating the difficulty (aka number of leading zero bits) in a nostr event id:

```
int zero_bits(unsigned char b)
{
        int n = 0;

        if (b == 0)
                return 8;

        while (b >>= 1)
                n++;

        return 7-n;
}

/* find the number of leading zero bits in a hash */
int count_leading_zero_bits(unsigned char *hash)
{
        int bits, total, i;
        for (i = 0, total = 0; i < 32; i++) {
                bits = zero_bits(hash[i]);
                total += bits;
                if (bits != 8)
                        break;
        }
        return total;
}
```

Here is some JavaScript code for doing the same thing:

```
// hex should be a hexadecimal string (with no 0x prefix)
function countLeadingZeroes(hex) {
  let count = 0;

  for (let i = 0; i < hex.length; i++) {
    const nibble = parseInt(hex[i], 16);
```

```
    if (nibble === 0) {
      count += 4;
    } else {
      count += Math.clz32(nibble) - 28;
      break;
    }
  }

  return count;
}
```

## Delegated Proof of Work

Since the `NIP-01` note id does not commit to any signature, PoW can be outsourced to PoW providers, perhaps for a fee. This provides a way for clients to get their messages out to PoW-restricted relays without having to do any work themselves, which is useful for energy-constrained devices like mobile phones.

## NIP-14

## Subject tag in Text events

`draft` `optional`

This NIP defines the use of the “subject” tag in text (kind: 1) events. (implemented in more-speech)

```
["subject": <string>]
```

Browsers often display threaded lists of messages. The contents of the subject tag can be used in such lists, instead of the more ad hoc approach of using the first few words of the message. This is very similar to the way email browsers display lists of incoming emails by subject rather than by contents.

When replying to a message with a subject, clients SHOULD replicate the subject tag. Clients MAY adorn the subject to denote that it is a reply. e.g. by prepending “Re:”.

Subjects should generally be shorter than 80 chars. Long subjects will likely be trimmed by clients.

## NIP-15

## Nostr Marketplace

`draft` `optional`

Based on Diagon-Alley .

Implemented in NostrMarket and Plebeian Market .

### Terms

- `merchant` - seller of products with NOSTR key-pair
- `customer` - buyer of products with NOSTR key-pair
- `product` - item for sale by the `merchant`
- `stall` - list of products controlled by `merchant` (a `merchant` can have multiple stalls)
- `marketplace` - clientside software for searching `stalls` and purchasing `products`

### Nostr Marketplace Clients

#### Merchant admin

Where the `merchant` creates, updates and deletes `stalls` and `products`, as well as where they manage sales, payments and communication with `customers`.

The `merchant` admin software can be purely clientside, but for `convenience` and uptime, implementations will likely have a server client listening for NOSTR events.

#### Marketplace

`Marketplace` software should be entirely clientside, either as a stand-alone app, or as a purely frontend webpage. A `customer` subscribes to different merchant NOSTR public keys, and those `merchants` `stalls` and `products` become listed and searchable. The marketplace client is like any other ecommerce site, with basket and checkout.

`Marketplaces` may also wish to include a `customer` support area for direct message communication with `merchants`.

### `Merchant` publishing/updating products (event)

A merchant can publish these events:

| Kind | | Description |
|---|---|---|
| 0 | `set_meta` | The merchant description (similar with any `nostr` public key). |
| 30017 | `set_stall` | Create or update a stall. |
| 30018 | `set_product` | Create or update a product. |
| 4 | `direct_message` | Communicate with the customer. The messages can be plain-text or JSON. |
| 5 | `delete` | Delete a product or a stall. |

#### Event `30017`: Create or update a stall.

#### Event Content

```
{
  "id": <string, id generated by the merchant. Sequential IDs (`0`, `1`, `2`...) are
discouraged>,
  "name": <string, stall name>,
  "description": <string (optional), stall description>,
  "currency": <string, currency used>,
  "shipping": [
    {
      "id": <string, id of the shipping zone, generated by the merchant>,
      "name": <string (optional), zone name>,
      "cost": <float, base cost for shipping. The currency is defined at the stall
level>,
      "regions": [<string, regions included in this zone>]
    }
  ]
}
```

Fields that are not self-explanatory:

- `shipping`:
  - an array with possible shipping zones for this stall.
  - the customer MUST choose exactly one of those shipping zones.
  - shipping to different zones can have different costs. For some goods (digital for example) the cost can be zero.
  - the `id` is an internal value used by the merchant. This value must be sent back as the customer selection.
  - each shipping zone contains the base cost for orders made to that shipping zone, but a specific shipping cost per product can also be specified if the shipping cost for that product is higher than what's specified by the base cost.

#### Event Tags

```
{
  "tags": [["d", <string, id of stall]],
  // other fields...
}
```

- the `d` tag is required, its value MUST be the same as the stall `id`.

#### Event `30018`: Create or update a product

#### Event Content

```
{
  "id": <string, id generated by the merchant (sequential ids are discouraged)>,
  "stall_id": <string, id of the stall to which this product belong to>,
  "name": <string, product name>,
  "description": <string (optional), product description>,
  "images": <[string], array of image URLs, optional>,
  "currency": <string, currency used>,
  "price": <float, cost of product>,
```

```
  "quantity": <int or null, available items>,
  "specs": [
    [<string, spec key>, <string, spec value>]
  ],
  "shipping": [
    {
      "id": <string, id of the shipping zone (must match one of the zones defined
for the stall)>,
      "cost": <float, extra cost for shipping. The currency is defined at the stall
level>
    }
  ]
}
```

Fields that are not self-explanatory:

- `quantity` can be null in the case of items with unlimited availability, like digital items, or services
- `specs`:
    - an optional array of key pair values. It allows for the Customer UI to present product specifications in a structure mode. It also allows comparison between products
    - eg: `[["operating_system", "Android 12.0"], ["screen_size", "6.4 inches"], ["connector_type", "USB Type C"]]`

  *Open*: better to move `spec` in the `tags` section of the event?
- `shipping`:
    - an *optional* array of extra costs to be used per shipping zone, only for products that require special shipping costs to be added to the base shipping cost defined in the stall
    - the `id` should match the id of the shipping zone, as defined in the `shipping` field of the stall
    - to calculate the total cost of shipping for an order, the user will choose a shipping option during checkout, and then the client must consider this costs:
        - the `base cost from the stall` for the chosen shipping option
        - the result of multiplying the product units by the `shipping costs specified in the product`, if any.

**Event Tags**

```
  "tags": [
    ["d", <string, id of product],
    ["t", <string (optional), product category],
    ["t", <string (optional), product category],
    // other fields...
  ],
  ...
```

- the `d` tag is required, its value MUST be the same as the product `id`.
- the `t` tag is as searchable tag, it represents different categories that the product can be part of (`food`, `fruits`). Multiple `t` tags can be present.

### Checkout events

All checkout events are sent as JSON strings using NIP-04 .

The `merchant` and the `customer` can exchange JSON messages that represent different actions. Each `JSON` message `MUST` have a `type` field indicating the what the JSON represents. Possible types:

| Message Type | Sent By | Description |
|---|---|---|
| 0 | Customer | New Order |
| 1 | Merchant | Payment Request |
| 2 | Merchant | Order Status Update |

**Step 1: `customer` order (event)**

The below JSON goes in content of NIP-04 .

```
{
  "id": <string, id generated by the customer>,
```

```
  "type": 0,
  "name": <string (optional), ???>,
  "address": <string (optional), for physical goods an address should be provided>,
  "message": <string (optional), message for merchant>,
  "contact": {
    "nostr": <32-bytes hex of a pubkey>,
    "phone": <string (optional), if the customer wants to be contacted by phone>,
    "email": <string (optional), if the customer wants to be contacted by email>
  },
  "items": [
    {
      "product_id": <string, id of the product>,
      "quantity": <int, how many products the customer is ordering>
    }
  ],
  "shipping_id": <string, id of the shipping zone>
}
```

*Open*: is `contact.nostr` required?

#### Step 2: `merchant` request payment (event)

Sent back from the merchant for payment. Any payment option is valid that the merchant can check.

The below JSON goes in `content` of NIP-04 .

`payment_options/type` include:

- `url` URL to a payment page, stripe, paypal, btcpayserver, etc
- `btc` onchain bitcoin address
- `ln` bitcoin lightning invoice
- `lnurl` bitcoin lnurl-pay

```
{
  "id": <string, id of the order>,
  "type": 1,
  "message": <string, message to customer, optional>,
  "payment_options": [
    {
      "type": <string, option type>,
      "link": <string, url, btc address, ln invoice, etc>
    },
    {
      "type": <string, option type>,
      "link": <string, url, btc address, ln invoice, etc>
    },
    {
      "type": <string, option type>,
      "link": <string, url, btc address, ln invoice, etc>
    }
  ]
}
```

#### Step 3: `merchant` verify payment/shipped (event)

Once payment has been received and processed.

The below JSON goes in `content` of NIP-04 .

```
{
  "id": <string, id of the order>,
  "type": 2,
  "message": <string, message to customer>,
  "paid": <bool: has received payment>,
  "shipped": <bool: has been shipped>,
}
```

### Customize Marketplace

Create a customized user experience using the `naddr` from NIP-19 . The use of `naddr` enables easy sharing of marketplace events while incorporating a rich set of metadata. This metadata can include relays, merchant profiles, and more. Subsequently, it allows merchants to be grouped into a market, empowering the market creator to configure the marketplace's user interface and user experience, and share that marketplace. This customization can encompass elements such as market name, description, logo, banner, themes, and even color schemes, offering a tailored and unique marketplace experience.

**Event `30019`: Create or update marketplace UI/UX**

**Event Content**

```
{
  "name": <string (optional), market name>,
  "about": <string (optional), market description>,
  "ui": {
    "picture": <string (optional), market logo image URL>,
    "banner": <string (optional), market logo banner URL>,
    "theme": <string (optional), market theme>,
    "darkMode": <bool, true/false>
  },
  "merchants": [array of pubkeys (optional)],
  // other fields...
}
```

This event leverages naddr to enable comprehensive customization and sharing of marketplace configurations, fostering a unique and engaging marketplace environment.

## Auctions

**Event `30020`: Create or update a product sold as an auction**

**Event Content**:

```
{
    "id": <String, UUID generated by the merchant. Sequential IDs (`0`, `1`, `2`...)
are discouraged>,
    "stall_id": <String, UUID of the stall to which this product belong to>,
    "name": <String, product name>,
    "description": <String (optional), product description>,
    "images": <[String], array of image URLs, optional>,
    "starting_bid": <int>,
    "start_date": <int (optional) UNIX timestamp, date the auction started / will
start>,
    "duration": <int, number of seconds the auction will run for, excluding eventual
time extensions that might happen>,
    "specs": [
        [<String, spec key>, <String, spec value>]
    ],
    "shipping": [
        {
            "id": <String, UUID of the shipping zone. Must match one of the zones
defined for the stall>,
            "cost": <float, extra cost for shipping. The currency is defined at the
stall level>
        }
    ]
}
```

> [!NOTE] Items sold as an auction are very similar in structure to fixed-price items, with some important differences worth noting.

- The `start_date` can be set to a date in the future if the auction is scheduled to start on that date, or can be omitted if the start date is unknown/hidden. If the start date is not specified, the auction will have to be edited later to set an actual date.
- The auction runs for an initial number of seconds after the `start_date`, specified by `duration`.

**Event `1021`: Bid**

```
{
    "content": <int, amount of sats>,
    "tags": [["e", <event ID of the auction to bid on>]],
    // other fields...
}
```

Bids are simply events of kind `1021` with a `content` field specifying the amount, in the currency of the auction. Bids must reference an auction.

> [!NOTE] Auctions can be edited as many times as desired (they are “addressable events”) by the author - even after the start_date, but they cannot be edited after they have received the first bid! This is enforced by the fact that bids reference the event ID of the auction (rather than the product UUID), which changes with every new version of the auctioned product. So a bid is always attached to one “version”. Editing the auction after a bid would result in the new product losing the bid!

**Event `1022`: Bid confirmation**

**Event Content**:

```
{
    "status": <String, "accepted" | "rejected" | "pending" | "winner">,
    "message": <String (optional)>,
    "duration_extended": <int (optional), number of seconds>
}
```

**Event Tags**:

```
  "tags": [["e" <event ID of the bid being confirmed>], ["e", <event ID of the auction>]],
```

Bids should be confirmed by the merchant before being considered as valid by other clients. So clients should subscribe to *bid confirmation* events (kind `1022`) for every auction that they follow, in addition to the actual bids and should check that the pubkey of the bid confirmation matches the pubkey of the merchant (in addition to checking the signature).

The `content` field is a JSON which includes *at least* a `status`. `winner` is how the *winning bid* is replied to after the auction ends and the winning bid is picked by the merchant.

The reasons for which a bid can be marked as `rejected` or `pending` are up to the merchant’s implementation and configuration - they could be anything from basic validation errors (amount too low) to the bidder being blacklisted or to the bidder lacking sufficient *trust*, which could lead to the bid being marked as `pending` until sufficient verification is performed. The difference between the two is that `pending` bids *might* get approved after additional steps are taken by the bidder, whereas `rejected` bids can not be later approved.

An additional `message` field can appear in the `content` JSON to give further context as of why a bid is `rejected` or `pending`.

Another thing that can happen is - if bids happen very close to the end date of the auction - for the merchant to decide to extend the auction duration for a few more minutes. This is done by passing a `duration_extended` field as part of a bid confirmation, which would contain a number of seconds by which the initial duration is extended. So the actual end date of an auction is always `start_date + duration + (SUM(c.duration_extended) FOR c in all confirmations`.

### Customer support events

Customer support is handled over whatever communication method was specified. If communicating via nostr, NIP-04 is used.

### Additional

Standard data models can be found here

## NIP-16

## Event Treatment

`final` `mandatory`

Moved to NIP-01 .

# NIP-17

## Private Direct Messages

`draft` `optional`

This NIP defines an encrypted direct messaging scheme using NIP-44 encryption and NIP-59 seals and gift wraps.

### Direct Message Kind

Kind `14` is a chat message. `p` tags identify one or more receivers of the message.

```
{
  "id": "<usual hash>",
  "pubkey": "<sender-pubkey>",
  "created_at": "<current-time>",
  "kind": 14,
  "tags": [
    ["p", "<receiver-1-pubkey>", "<relay-url>"],
    ["p", "<receiver-2-pubkey>", "<relay-url>"],
    ["e", "<kind-14-id>", "<relay-url>", "reply"] // if this is a reply
    ["subject", "<conversation-title>"],
    // rest of tags...
  ],
  "content": "<message-in-plain-text>",
}
```

`.content` MUST be plain text. Fields `id` and `created_at` are required.

Tags that mention, quote and assemble threading structures MUST follow NIP-10 .

Kind `14`s MUST never be signed. If it is signed, the message might leak to relays and become **fully public**.

### Chat Rooms

The set of `pubkey` + `p` tags defines a chat room. If a new `p` tag is added or a current one is removed, a new room is created with clean message history.

Clients SHOULD render messages of the same room in a continuous thread.

An optional `subject` tag defines the current name/topic of the conversation. Any member can change the topic by simply submitting a new `subject` to an existing `pubkey` + `p`-tags room. There is no need to send `subject` in every message. The newest `subject` in the thread is the subject of the conversation.

### Encrypting

Following NIP-59 , the **unsigned** `kind:14` chat message must be sealed (`kind:13`) and then gift-wrapped (`kind:1059`) to each receiver and the sender individually.

```
{
  "id": "<usual hash>",
  "pubkey": randomPublicKey,
  "created_at": randomTimeUpTo2DaysInThePast(),
  "kind": 1059, // gift wrap
  "tags": [
    ["p", receiverPublicKey, "<relay-url>"] // receiver
  ],
  "content": nip44Encrypt(
    {
      "id": "<usual hash>",
      "pubkey": senderPublicKey,
      "created_at": randomTimeUpTo2DaysInThePast(),
      "kind": 13, // seal
      "tags": [], // no tags
      "content": nip44Encrypt(unsignedKind14, senderPrivateKey, receiverPublicKey),
      "sig": "<signed by senderPrivateKey>"
    },
    randomPrivateKey, receiverPublicKey
  ),
```

```
  "sig": "<signed by randomPrivateKey>"
}
```

The encryption algorithm MUST use the latest version of NIP-44 .

Clients MUST verify if pubkey of the `kind:13` is the same pubkey on the `kind:14`, otherwise any sender can impersonate others by simply changing the pubkey on `kind:14`.

Clients SHOULD randomize `created_at` in up to two days in the past in both the seal and the gift wrap to make sure grouping by `created_at` doesn't reveal any metadata.

The gift wrap's `p`-tag can be the receiver's main pubkey or an alias key created to receive DMs without exposing the receiver's identity.

Clients CAN offer disappearing messages by setting an `expiration` tag in the gift wrap of each receiver or by not generating a gift wrap to the sender's public key

### Publishing

Kind `10050` indicates the user's preferred relays to receive DMs. The event MUST include a list of `relay` tags with relay URIs.

```
{
  "kind": 10050,
  "tags": [
    ["relay", "wss://inbox.nostr.wine"],
    ["relay", "wss://myrelay.nostr1.com"],
  ],
  "content": "",
  // other fields...
}
```

Clients SHOULD publish kind `14` events to the `10050`-listed relays. If that is not found that indicates the user is not ready to receive messages under this NIP and clients shouldn't try.

### Relays

It's advisable that relays do not serve `kind:1059` to clients other than the ones tagged in them.

It's advisable that users choose relays that conform to these practices.

Clients SHOULD guide users to keep `kind:10050` lists small (1-3 relays) and SHOULD spread it to as many relays as viable.

### Benefits & Limitations

This NIP offers the following privacy and security features:

1. **No Metadata Leak**: Participant identities, each message's real date and time, event kinds, and other event tags are all hidden from the public. Senders and receivers cannot be linked with public information alone.
2. **No Public Group Identifiers**: There is no public central queue, channel or otherwise converging identifier to correlate or count all messages in the same group.
3. **No Moderation**: There are no group admins: no invitations or bans.
4. **No Shared Secrets**: No secret must be known to all members that can leak or be mistakenly shared
5. **Fully Recoverable**: Messages can be fully recoverable by any client with the user's private key
6. **Optional Forward Secrecy**: Users and clients can opt-in for "disappearing messages".
7. **Uses Public Relays**: Messages can flow through public relays without loss of privacy. Private relays can increase privacy further, but they are not required.
8. **Cold Storage**: Users can unilaterally opt-in to sharing their messages with a separate key that is exclusive for DM backup and recovery.

The main limitation of this approach is having to send a separate encrypted event to each receiver. Group chats with more than 100 participants should find a more suitable messaging scheme.

### Implementation

Clients implementing this NIP should by default only connect to the set of relays found in their `kind:10050` list. From that they should be able to load all messages both sent and received as well as get new live updates, making it for a very simple and lightweight implementation that should be fast.

When sending a message to anyone, clients must then connect to the relays in the receiver's `kind:10050` and send the events there, but can disconnect right after unless more messages are expected to be sent (e.g. the chat tab is still selected). Clients should also send a copy of their outgoing messages to their own `kind:10050` relay set.

### Examples

This example sends the message `Hola, que tal?` from `nsec1w8udu59ydjvedgs3yv5qccshcj8k05fh3l60k9x57asjrqdpa00qkmr89m` to `nsec12ywtkplvyq5t6twdqwwygavp5lm4fhuang89c943nf2z92eez43szvn4dt`.

The two final GiftWraps, one to the receiver and the other to the sender, respectively, are:


```
{
   "id":"2886780f7349afc1344047524540ee716f7bdc1b64191699855662330bf235d8",
   "pubkey":"8f8a7ec43b77d25799281207e1a47f7a654755055788f7482653f9c9661c6d51",
   "created_at":1703128320,
   "kind":1059,
   "tags":[
      [ "p", "918e2da906df4ccd12c8ac672d8335add131a4cf9d27ce42b3bb3625755f0788"]
   ],

"content":"AsqzdlMsG304G8h08bE67dhAR1gFTzTckUUyuvndZ8LrGCvwI4pgC3d6hyAK0Wo9gtkLqSr2rT2RyHlE5wRqbCOlQ8W


"sig":"a3c6ce632b145c0869423c1afaff4a6d764a9b64dedaf15f170b944ead67227518a72e455567ca1c2a0d187832cecbd

}
```



```
{
   "id":"162b0611a1911cfcb30f8a5502792b346e535a45658b3a31ae5c178465509721",
   "pubkey":"626be2af274b29ea4816ad672ee452b7cf96bbb4836815a55699ae402183f512",
   "created_at":1702711587,
   "kind":1059,
   "tags":[
      [ "p", "44900586091b284416a0c001f677f9c49f7639a55c3f1e2ec130a8e1a7998e1b"]
   ],

"content":"AsTClTzr0gzXXji7uye5UB6LYrx3HDjWGdkNaBS6BAX9CpHa+Vvtt5oI2xJrmWLen+Fo2NBOFazvl285Gb3HSM82gVy


"sig":"c94e74533b482aa8eeeb54ae72a5303e0b21f62909ca43c8ef06b0357412d6f8a92f96e1a205102753777fd25321a58

}
```


## NIP-18

## Reposts

`draft` `optional`

A repost is a `kind 6` event that is used to signal to followers that a `kind 1` text note is worth reading.

The `content` of a repost event is *the stringified JSON of the reposted note*. It MAY also be empty, but that is not recommended.

The repost event MUST include an `e` tag with the `id` of the note that is being reposted. That tag MUST include a relay URL as its third entry to indicate where it can be fetched.

The repost SHOULD include a `p` tag with the `pubkey` of the event being reposted.

### Quote Reposts

Quote reposts are `kind 1` events with an embedded `q` tag of the note being quote reposted. The `q` tag ensures quote reposts are not pulled and included as replies in threads. It also allows you to easily pull and count all of the quotes for a post.

`q` tags should follow the same conventions as NIP 10 `e` tags, with the exception of the `mark` argument.

```
["q", <event-id>, <relay-url>, <pubkey>]
```

Quote reposts MUST include the NIP-21 `nevent`, `note`, or `naddr` of the event in the content.

### Generic Reposts

Since `kind 6` reposts are reserved for `kind 1` contents, we use `kind 16` as a "generic repost", that can include any kind of event inside other than `kind 1`.

`kind 16` reposts SHOULD contain a `k` tag with the stringified kind number of the reposted event as its value.

## NIP-19

## bech32-encoded entities

`draft` `optional`

This NIP standardizes bech32-formatted strings that can be used to display keys, ids and other information in clients. These formats are not meant to be used anywhere in the core protocol, they are only meant for displaying to users, copy-pasting, sharing, rendering QR codes and inputting data.

It is recommended that ids and keys are stored in either hex or binary format, since these formats are closer to what must actually be used the core protocol.

### Bare keys and ids

To prevent confusion and mixing between private keys, public keys and event ids, which are all 32 byte strings. bech32-(not-m) encoding with different prefixes can be used for each of these entities.

These are the possible bech32 prefixes:

- `npub`: public keys
- `nsec`: private keys
- `note`: note ids

Example: the hex public key `3bf0c63fcb93463407af97a5e5ee64fa883d107ef9e558472c4eb9aaaefa459d` translates to `npub180cvv07tjdrrgpa0j7j7tmnyl2yr6yr7l8j4s3evf6u64th6gkwsyjh6w6`.

The bech32 encodings of keys and ids are not meant to be used inside the standard NIP-01 event formats or inside the filters, they're meant for human-friendlier display and input only. Clients should still accept keys in both hex and npub format for now, and convert internally.

### Shareable identifiers with extra metadata

When sharing a profile or an event, an app may decide to include relay information and other metadata such that other apps can locate and display these entities more easily.

For these events, the contents are a binary-encoded list of `TLV` (type-length-value), with `T` and `L` being 1 byte each (`uint8`, i.e. a number in the range of 0-255), and `V` being a sequence of bytes of the size indicated by `L`.

These are the possible bech32 prefixes with `TLV`:

- `nprofile`: a nostr profile
- `nevent`: a nostr event
- `naddr`: a nostr *replaceable event* coordinate
- `nrelay`: a nostr relay (deprecated)

These possible standardized `TLV` types are indicated here:

- `0`: `special`
  - depends on the bech32 prefix:
    - for `nprofile` it will be the 32 bytes of the profile public key
    - for `nevent` it will be the 32 bytes of the event id
    - for `naddr`, it is the identifier (the `"d"` tag) of the event being referenced. For normal replaceable events use an empty string.
- `1`: `relay`
  - for `nprofile`, `nevent` and `naddr`, *optionally*, a relay in which the entity (profile or event) is more likely to be found, encoded as ascii
  - this may be included multiple times
- `2`: `author`
  - for `naddr`, the 32 bytes of the pubkey of the event
  - for `nevent`, *optionally*, the 32 bytes of the pubkey of the event

- 3: `kind`
  - for `naddr`, the 32-bit unsigned integer of the kind, big-endian
  - for `nevent`, *optionally*, the 32-bit unsigned integer of the kind, big-endian

### Examples

- `npub10elfcs4fr0l0r8af98jlmgdh9c8tcxjvz9qkw038js35mp4dma8qzvjptg` should decode into the public key hex `7e7e9c42a91bfef19fa929e5fda1b72e0ebc1a4c1141673e2794234d86addf4e` and vice-versa
- `nsec1vl029mgpspedva04g90vltkh6fvh240zqtv9k0t9af8935ke9laqsnlfe5` should decode into the private key hex `67dea2ed018072d675f5415ecfaed7d2597555e202d85b3d65ea4e58d2d92ffa` and vice-versa
- `nprofile1qqsrhuxx8l9ex335q7he0f09aej04zpazpl0ne2cgukyawd24mayt8gpp4mhxue69uhhytnc9e3k7mgpz4mhxue69u` should decode into a profile with the following TLV items:
  - pubkey: `3bf0c63fcb93463407af97a5e5ee64fa883d107ef9e558472c4eb9aaaefa459d`
  - relay: `wss://r.x.com`
  - relay: `wss://djbas.sadkb.com`

### Notes

- `npub` keys MUST NOT be used in NIP-01 events or in NIP-05 JSON responses, only the hex format is supported there.
- When decoding a bech32-formatted string, TLVs that are not recognized or supported should be ignored, rather than causing an error.

## NIP-20

## Command Results

`final` `mandatory`

Moved to NIP-01 .

## NIP-21

## `nostr:` URI scheme

`draft` `optional`

This NIP standardizes the usage of a common URI scheme for maximum interoperability and openness in the network.

The scheme is `nostr:`.

The identifiers that come after are expected to be the same as those defined in NIP-19 (except `nsec`).

### Examples

- `nostr:npub1sn0wdenkukak0d9dfczzeacvhkrgz92ak56egt7vdgzn8pv2wfqqhrjdv9`
- `nostr:nprofile1qqsrhuxx8l9ex335q7he0f09aej04zpazpl0ne2cgukyawd24mayt8gpp4mhxue69uhhytnc9e3k7mgpz4mh`
- `nostr:note1fntxtkcy9pjwucqwa9mddn7v03wwwsu9j330jj350nvhpky2tuaspk6nqc`
- `nostr:nevent1qqstna2yrezu5wghjvswqqculvvwxsrcvu7uc0f78gan4xqhvz49d9spr3mhxue69uhkummnw3ez6un9d3shjt`

## NIP-22

## Comment

`draft` `optional`

A comment is a threading note always scoped to a root event or an `I`-tag.

It uses `kind:1111` with plaintext `.content` (no HTML, Markdown, or other formatting).

Comments MUST point to the root scope using uppercase tag names (e.g. `K`, `E`, `A` or `I`) and MUST point to the parent item with lowercase ones (e.g. `k`, `e`, `a` or `i`).

Comments MUST point to the authors when one is available (i.e. tagging a nostr event). `P` for the root scope and `p` for the author of the parent item.

```
{
  kind: 1111,
  content: '<comment>',
  tags: [
    // root scope: event addresses, event ids, or I-tags.
    ["<A, E, I>", "<address, id or I-value>", "<relay or web page hint>", "<root
event's pubkey, if an E tag>"],
    // the root item kind
    ["K", "<root kind>"],

    // pubkey of the author of the root scope event
    ["P", "<root-pubkey>", "relay-url-hint"],

    // parent item: event addresses, event ids, or i-tags.
    ["<a, e, i>", "<address, id or i-value>", "<relay or web page hint>", "<parent
event's pubkey, if an e tag>"],
    // parent item kind
    ["k", "<parent comment kind>"],

    // parent item pubkey
    ["p", "<parent-pubkey>", "relay-url-hint"]
  ]
  // other fields
}
```

Tags `K` and `k` MUST be present to define the event kind of the root and the parent items.

`I` and `i` tags create scopes for hashtags, geohashes, URLs, and other external identifiers.

The possible values for `i` tags – and `k` tags, when related to an extenal identity – are listed on NIP-73 . Their uppercase versions use the same type of values but relate to the root item instead of the parent one.

`q` tags MAY be used when citing events in the `.content` with NIP-21 .

```
["q", "<event-id> or <event-address>", "<relay-url>", "<pubkey-if-a-regular-event>"]
```

`p` tags SHOULD be used when mentioning pubkeys in the `.content` with NIP-21 .

### Examples

A comment on a blog post looks like this:

```
{
  kind: 1111,
  content: 'Great blog post!',
  tags: [
    // top-level comments scope to event addresses or ids
    ["A",
"30023:3c9849383bdea883b0bd16fece1ed36d37e37cdde3ce43b17ea4e9192ec11289:f9347ca7",
"wss://example.relay"],
    // the root kind
    ["K", "30023"],
    // author of root event
    ["P", "3c9849383bdea883b0bd16fece1ed36d37e37cdde3ce43b17ea4e9192ec11289",
"wss://example.relay"]

    // the parent event address (same as root for top-level comments)
    ["a",
"30023:3c9849383bdea883b0bd16fece1ed36d37e37cdde3ce43b17ea4e9192ec11289:f9347ca7",
"wss://example.relay"],
    // when the parent event is replaceable or addressable, also include an `e` tag
referencing its id
    ["e", "5b4fc7fed15672fefe65d2426f67197b71ccc82aa0cc8a9e94f683eb78e07651",
"wss://example.relay"],
    // the parent event kind
    ["k", "30023"],
    // author of the parent event
    ["p", "3c9849383bdea883b0bd16fece1ed36d37e37cdde3ce43b17ea4e9192ec11289",
```

```
"wss://example.relay"]
  ]
  // other fields
}
```

A comment on a NIP-94 file looks like this:

```
{
  kind: 1111,
  content: 'Great file!',
  tags: [
    // top-level comments have the same scope and reply to addresses or ids
    ["E", "768ac8720cdeb59227cf95e98b66560ef03d8bc9a90d721779e76e68fb42f5e6",
"wss://example.relay",
"3721e07b079525289877c366ccab47112bdff3d1b44758ca333feb2dbbbbe5bb"],
    // the root kind
    ["K", "1063"],
    // author of the root event
    ["P", "3721e07b079525289877c366ccab47112bdff3d1b44758ca333feb2dbbbbe5bb"],

    // the parent event id (same as root for top-level comments)
    ["e", "768ac8720cdeb59227cf95e98b66560ef03d8bc9a90d721779e76e68fb42f5e6",
"wss://example.relay",
"3721e07b079525289877c366ccab47112bdff3d1b44758ca333feb2dbbbbe5bb"],
    // the parent kind
    ["k", "1063"],
    ["p", "3721e07b079525289877c366ccab47112bdff3d1b44758ca333feb2dbbbbe5bb"]
  ]
  // other fields
}
```

A reply to a comment looks like this:

```
{
  kind: 1111,
  content: 'This is a reply to "Great file!"',
  tags: [
    // nip-94 file event id
    ["E", "768ac8720cdeb59227cf95e98b66560ef03d8bc9a90d721779e76e68fb42f5e6",
"wss://example.relay",
"fd913cd6fa9edb8405750cd02a8bbe16e158b8676c0e69fdc27436cc4a54cc9a"],
    // the root kind
    ["K", "1063"],
    ["P", "fd913cd6fa9edb8405750cd02a8bbe16e158b8676c0e69fdc27436cc4a54cc9a"],

    // the parent event
    ["e", "5c83da77af1dec6d7289834998ad7aafbd9e2191396d75ec3cc27f5a77226f36",
"wss://example.relay",
"93ef2ebaaf9554661f33e79949007900bbc535d239a4c801c33a4d67d3e7f546"],
    // the parent kind
    ["k", "1111"],
    ["p", "93ef2ebaaf9554661f33e79949007900bbc535d239a4c801c33a4d67d3e7f546"]
  ]
  // other fields
}
```

A comment on a website's url looks like this:

```
{
  kind: 1111,
  content: 'Nice article!',
  tags: [
    // referencing the root url
    ["I", "https://abc.com/articles/1"],
    // the root "kind": for an url, the kind is its domain
    ["K", "https://abc.com"],

    // the parent reference (same as root for top-level comments)
```

```
    ["i", "https://abc.com/articles/1"],
    // the parent "kind": for an url, the kind is its domain
    ["k", "https://abc.com"]
  ]
  // other fields
}
```

A podcast comment example:

```
{
  id: "80c48d992a38f9c445b943a9c9f1010b3966760134437657504 31a9004bdac05",
  pubkey: "252f10c83610ebca1a059c0bae8255eba2f95be4d1d7bcfa89d7248a82d9f111",
  kind: 1111,
  content: "This was a great episode!",
  tags: [
    // podcast episode reference
    ["I", "podcast:item:guid:d98d189b-dc7b-45b1-8720-d4b98690f31f",
"https://fountain.fm/episode/z1y9TMQRuqXl2awyrQxg"],
    // podcast episode type
    ["K", "podcast:item:guid"],

    // same value as "I" tag above, because it is a top-level comment (not a reply
to a comment)
    ["i", "podcast:item:guid:d98d189b-dc7b-45b1-8720-d4b98690f31f",
"https://fountain.fm/episode/z1y9TMQRuqXl2awyrQxg"],
    ["k", "podcast:item:guid"]
  ]
  // other fields
}
```

A reply to a podcast comment:

```
{
  kind: 1111,
  content: "I'm replying to the above comment.",
  tags: [
    // podcast episode reference
    ["I", "podcast:item:guid:d98d189b-dc7b-45b1-8720-d4b98690f31f",
"https://fountain.fm/episode/z1y9TMQRuqXl2awyrQxg"],
    // podcast episode type
    ["K", "podcast:item:guid"],

    // this is a reference to the above comment
    ["e", "80c48d992a38f9c445b943a9c9f1010b3966760134437657504 31a9004bdac05",
"wss://example.relay",
"252f10c83610ebca1a059c0bae8255eba2f95be4d1d7bcfa89d7248a82d9f111"],
    // the parent comment kind
    ["k", "1111"]
    ["p", "252f10c83610ebca1a059c0bae8255eba2f95be4d1d7bcfa89d7248a82d9f111"]
  ]
  // other fields
}
```

## NIP-23

## Long-form Content

`draft` `optional`

This NIP defines `kind:30023` (an *addressable event*) for long-form text content, generally referred to as “articles” or “blog posts”. `kind:30024` has the same structure as `kind:30023` and is used to save long form drafts.

“Social” clients that deal primarily with `kind:1` notes should not be expected to implement this NIP.

#### Format

The `.content` of these events should be a string text in Markdown syntax. To maximize compatibility and readability between different clients and devices, any client that is creating long form notes:

- MUST NOT hard line-break paragraphs of text, such as arbitrary line breaks at 80 column boundaries.
- MUST NOT support adding HTML to Markdown.

**Metadata**

For the date of the last update the `.created_at` field should be used, for “tags”/“hashtags” (i.e. topics about which the event might be of relevance) the `t` tag should be used.

Other metadata fields can be added as tags to the event as necessary. Here we standardize 4 that may be useful, although they remain strictly optional:

- `"title"`, for the article title
- `"image"`, for a URL pointing to an image to be shown along with the title
- `"summary"`, for the article summary
- `"published_at"`, for the timestamp in unix seconds (stringified) of the first time the article was published

**Editability**

These articles are meant to be editable, so they should include a `d` tag with an identifier for the article. Clients should take care to only publish and read these events from relays that implement that. If they don’t do that they should also take care to hide old versions of the same article they may receive.

**Linking**

The article may be linked to using the NIP-19 `naddr` code along with the `a` tag.

**References**

References to other Nostr notes, articles or profiles must be made according to NIP-27 , i.e. by using NIP-21 `nostr:...` links and optionally adding tags for these (see example below).

### Example Event

```
{
  "kind": 30023,
  "created_at": 1675642635,
  "content": "Lorem [ipsum]
[nostr:nevent1qqst8cujky046negxgwwm5ynqwn53t8aqjr6afd8g59nfqwxpdhylpcpzamhxue69uhhyetvv9ujuetcv9khqmr9
 dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut
labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation
ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in
reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.
Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt
mollit anim id est laborum.\n\nRead more at
nostr:naddr1qqzkjurnw4ksz9thwden5te0wfjkccte9ehx7um5wghx7un8qgs2d90kkcq3nk2jry62dyf50k0h36rhpdtd594my4

  "tags": [
    ["d", "lorem-ipsum"],
    ["title", "Lorem Ipsum"],
    ["published_at", "1296962229"],
    ["t", "placeholder"],
    ["e", "b3e392b11f5d4f28321cedd09303a748acfd0487aea5a7450b3481c60b6e4f87",
"wss://relay.example.com"],
    ["a",
"30023:a695f6b60119d9521934a691347d9f78e8770b56da16bb255ee286ddf9fda919:ipsum",
"wss://relay.nostr.org"]
  ],
  "pubkey": "...",
  "id": "..."
}
```

## NIP-24

## Extra metadata fields and tags

`draft` `optional`

This NIP keeps track of extra optional fields that can added to events which are not defined anywhere else but have become *de facto* standards and other minor implementation possibilities that do not deserve their own NIP and do not have a place in other NIPs.

## kind 0

These are extra fields not specified in NIP-01 that may be present in the stringified JSON of metadata events:

- `display_name`: an alternative, bigger name with richer characters than `name`. `name` should always be set regardless of the presence of `display_name` in the metadata.
- `website`: a web URL related in any way to the event author.
- `banner`: an URL to a wide (~1024x768) picture to be optionally displayed in the background of a profile screen.
- `bot`: a boolean to clarify that the content is entirely or partially the result of automation, such as with chatbots or newsfeeds.

#### Deprecated fields

These are fields that should be ignored or removed when found in the wild:

- `displayName`: use `display_name` instead.
- `username`: use `name` instead.

## kind 3

These are extra fields not specified in NIP-02 that may be present in the stringified JSON of follow events:

#### Deprecated fields

- `{<relay-url>: {"read": <true|false>, "write": <true|false>}, ...}`: an object of relays used by a user to read/write. NIP-65 should be used instead.

## tags

These tags may be present in multiple event kinds. Whenever a different meaning is not specified by some more specific NIP, they have the following meanings:

- `r`: a web URL the event is referring to in some way.
- `i`: an external id the event is referring to in some way - see NIP-73 .
- `title`: name of NIP-51 sets, NIP-52 calendar event, NIP-53 live event or NIP-99 listing.
- `t`: a hashtag. The value MUST be a lowercase string.

## NIP-25

## Reactions

`draft` `optional`

A reaction is a `kind 7` event that is used to react to other events.

The generic reaction, represented by the `content` set to a `+` string, SHOULD be interpreted as a “like” or “upvote”.

A reaction with `content` set to `-` SHOULD be interpreted as a “dislike” or “downvote”. It SHOULD NOT be counted as a “like”, and MAY be displayed as a downvote or dislike on a post. A client MAY also choose to tally likes against dislikes in a reddit-like system of upvotes and downvotes, or display them as separate tallies.

The `content` MAY be an emoji, or NIP-30 custom emoji in this case it MAY be interpreted as a “like” or “dislike”, or the client MAY display this emoji reaction on the post. If the `content` is an empty string then the client should consider it a “+”.

## Tags

The reaction event SHOULD include `e` and `p` tags from the note the user is reacting to (and optionally `a` tags if the target is a replaceable event). This allows users to be notified of reactions to posts they were mentioned in. Including the `e` tags enables clients to pull all the reactions associated with individual posts or all the posts in a thread. `a` tags enables clients to seek reactions for all versions of a replaceable event.

The last `e` tag MUST be the `id` of the note that is being reacted to.

The last `p` tag MUST be the `pubkey` of the event being reacted to.

The `a` tag MUST contain the coordinates (`kind:pubkey:d-tag`) of the replaceable being reacted to.

The reaction event MAY include a `k` tag with the stringified kind number of the reacted event as its value.

Example code

```
func make_like_event(pubkey: String, privkey: String, liked: NostrEvent) ->
NostrEvent {
    var tags: [[String]] = liked.tags.filter {
        tag in tag.count >= 2 && (tag[0] == "e" || tag[0] == "p")
    }
    tags.append(["e", liked.id])
    tags.append(["p", liked.pubkey])
    tags.append(["k", liked.kind])
    let ev = NostrEvent(content: "+", pubkey: pubkey, kind: 7, tags: tags)
    ev.calculate_id()
    ev.sign(privkey: privkey)
    return ev
}
```

## Reactions to a website

If the target of the reaction is a website, the reaction MUST be a `kind 17` event and MUST include an `r` tag with the website's URL.

```
{
  "kind": 17,
  "content": "⭐",
  "tags": [
    ["r", "https://example.com/"]
  ],
  // other fields...
}
```

URLs SHOULD be normalized , so that reactions to the same website are not omitted from queries. A fragment MAY be attached to the URL, to react to a section of the page. It should be noted that a URL with a fragment is not considered to be the same URL as the original.

## Custom Emoji Reaction

The client may specify a custom emoji (NIP-30 ) `:shortcode:` in the reaction content. The client should refer to the emoji tag and render the content as an emoji if shortcode is specified.

```
{
  "kind": 7,
  "content": ":soapbox:",
  "tags": [
    ["emoji", "soapbox", "https://gleasonator.com/emoji/Gleasonator/soapbox.png"]
  ],
  // other fields...
}
```

The content can be set only one `:shortcode:`. And emoji tag should be one.

## NIP-26

## Delegated Event Signing

`draft` `optional`

This NIP defines how events can be delegated so that they can be signed by other keypairs.

Another application of this proposal is to abstract away the use of the 'root' keypairs when interacting with clients. For example, a user could generate new keypairs for each client they wish to use and authorize those keypairs to generate events on behalf of their root pubkey, where the root keypair is stored in cold storage.

#### Introducing the 'delegation' tag

This NIP introduces a new tag: `delegation` which is formatted as follows:

```
[
  "delegation",
  <pubkey of the delegator>,
  <conditions query string>,
  <delegation token: 64-byte Schnorr signature of the sha256 hash of the delegation
string>
]
```

#### Delegation Token

The **delegation token** should be a 64-byte Schnorr signature of the sha256 hash of the following string:

```
nostr:delegation:<pubkey of publisher (delegatee)>:<conditions query string>
```

#### Conditions Query String

The following fields and operators are supported in the above query string:

*Fields*:

1. `kind`
    - *Operators*:
        - `=${KIND_NUMBER}` - delegatee may only sign events of this kind
2. `created_at`
    - *Operators*:
        - `<${TIMESTAMP}` - delegatee may only sign events created ***before*** the specified timestamp
        - `>${TIMESTAMP}` - delegatee may only sign events created ***after*** the specified timestamp

In order to create a single condition, you must use a supported field and operator. Multiple conditions can be used in a single query string, including on the same field. Conditions must be combined with `&`.

For example, the following condition strings are valid:

- `kind=1&created_at<1675721813`
- `kind=0&kind=1&created_at>1675721813`
- `kind=1&created_at>1674777689&created_at<1675721813`

For the vast majority of use-cases, it is advisable that:

1. Query strings should include a `created_at` ***after*** condition reflecting the current time, to prevent the delegatee from publishing historic notes on the delegator's behalf.
2. Query strings should include a `created_at` ***before*** condition that is not empty and is not some extremely distant time in the future. If delegations are not limited in time scope, they expose similar security risks to simply using the root key for authentication.

#### Example

```
## Delegator:
privkey: ee35e8bb71131c02c1d7e73231daa48e9953d329a4b701f7133c8f46dd21139c
pubkey:  8e0d3d3eb2881ec137a11debe736a9086715a8c8beeeda615780064d68bc25dd

## Delegatee:
privkey: 777e4f60b4aa87937e13acc84f7abcc3c93cc035cb4c1e9f7a9086dd78fffce1
pubkey:  477318cfb5427b9cfc66a9fa376150c1ddbc62115ae27cef72417eb959691396
```

Delegation string to grant note publishing authorization to the delegatee (477318cf) from now, for the next 30 days, given the current timestamp is `1674834236`.

```
nostr:delegation:477318cfb5427b9cfc66a9fa376150c1ddbc62115ae27cef72417eb959691396:kind=1&created_at>16
```

The delegator (8e0d3d3e) then signs a SHA256 hash of the above delegation string, the result of which is the delegation token:

```
6f44d7fe4f1c09f3954640fb58bd12bae8bb8ff4120853c4693106c82e920e2b898f1f9ba9bd65449a987c39c0423426ab7b53
```

The delegatee (477318cf) can now construct an event on behalf of the delegator (8e0d3d3e). The delegatee then signs the event with its own private key and publishes.

```
{
  "id": "e93c6095c3db1c31d15ac771f8fc5fb672f6e52cd25505099f62cd055523224f",
  "pubkey": "477318cfb5427b9cfc66a9fa376150c1ddbc62115ae27cef72417eb959691396",
  "created_at": 1677426298,
  "kind": 1,
  "tags": [
    [
      "delegation",
      "8e0d3d3eb2881ec137a11debe736a9086715a8c8beeeda615780064d68bc25dd",
      "kind=1&created_at>1674834236&created_at<1677426236",

"6f44d7fe4f1c09f3954640fb58bd12bae8bb8ff4120853c4693106c82e920e2b898f1f9ba9bd65449a987c39c0423426ab7b5

    ]
  ],
  "content": "Hello, world!",
  "sig":
"633db60e2e7082c13a47a6b19d663d45b2a2ebdeaf0b4c35ef83be2738030c54fc7fd56d139652937cdca875ee61b51904a1d

}
```

The event should be considered a valid delegation if the conditions are satisfied (`kind=1`, `created_at>1674834236` and `created_at<1677426236` in this example) and, upon validation of the delegation token, are found to be unchanged from the conditions in the original delegation string.

Clients should display the delegated note as if it was published directly by the delegator (8e0d3d3e).

#### Relay & Client Support

Relays should answer requests such as `["REQ", "", {"authors": ["A"]}]` by querying both the `pubkey` and delegation tags `[1]` value.

Relays SHOULD allow the delegator (8e0d3d3e) to delete the events published by the delegatee (477318cf).

## NIP-27

## Text Note References

`draft` `optional`

This document standardizes the treatment given by clients of inline references of other events and profiles inside the `.content` of any event that has readable text in its `.content` (such as kinds 1 and 30023).

When creating an event, clients should include mentions to other profiles and to other events in the middle of the `.content` using NIP-21 codes, such as `nostr:nprofile1qqsw3dy8cpu...6x2argwghx6egsqstvg`.

Including NIP-10 -style tags (`["e", <hex-id>, <relay-url>, <marker>]`) for each reference is optional, clients should do it whenever they want the profile being mentioned to be notified of the mention, or when they want the referenced event to recognize their mention as a reply.

A reader client that receives an event with such `nostr:...` mentions in its `.content` can do any desired context augmentation (for example, linking to the profile or showing a preview of the mentioned event contents) it wants in the process. If turning such mentions into links, they could become internal links, NIP-21 links or direct links to web clients that will handle these references.

---

### Example of a profile mention process

Suppose Bob is writing a note in a client that has search-and-autocomplete functionality for users that is triggered when they write the character `@`.

As Bob types `"hello @mat"` the client will prompt him to autocomplete with mattn's profile , showing a picture and name.

Bob presses "enter" and now he sees his typed note as `"hello @mattn"`, `@mattn` is highlighted, indicating that it is a mention. Internally, however, the event looks like this:

```
{
  "content": "hello
nostr:nprofile1qqszclxx9f5haga8sfjjrulaxncvkfekj097t6f3pu65f86rvg49ehqj6f9dh",
  "created_at": 1679790774,
  "id": "f39e9b451a73d62abc5016cffdd294b1a904e2f34536a208874fe5e22bbd47cf",
  "kind": 1,
  "pubkey": "79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798",
  "sig":
"f8c8bab1b90cc3d2ae1ad999e6af8af449ad8bb4edf64807386493163e29162b5852a796a8f474d6b1001cddbaac0de439283

  "tags": [
    [
      "p",
      "2c7cc62a697ea3a7826521f3fd34f0cb273693cbe5e9310f35449f43622a5cdc"
    ]
  ]
}
```


(Alternatively, the mention could have been a `nostr:npub1...` URL.)

After Bob publishes this event and Carol sees it, her client will initially display the `.content` as it is, but later it will parse the `.content` and see that there is a `nostr:` URL in there, decode it, extract the public key from it (and possibly relay hints), fetch that profile from its internal database or relays, then replace the full URL with the name `@mattn`, with a link to the internal page view for that profile.

### Verbose and probably unnecessary considerations

- The example above was very concrete, but it doesn't mean all clients have to implement the same flow. There could be clients that do not support autocomplete at all, so they just allow users to paste raw NIP-19 codes into the body of text, then prefix these with `nostr:` before publishing the event.
- The flow for referencing other events is similar: a user could paste a `note1...` or `nevent1...` code and the client will turn that into a `nostr:note1...` or `nostr:nevent1...` URL. Then upon reading such references the client may show the referenced note in a preview box or something like that – or nothing at all.
- Other display procedures can be employed: for example, if a client that is designed for dealing with only `kind:1` text notes sees, for example, a `kind:30023` `nostr:naddr1...` URL reference in the `.content`, it can, for example, decide to turn that into a link to some hardcoded webapp capable of displaying such events.
- Clients may give the user the option to include or not include tags for mentioned events or profiles. If someone wants to mention `mattn` without notifying them, but still have a nice augmentable/clickable link to their profile inside their note, they can instruct their client to *not* create a `["p", ...]` tag for that specific mention.
- In the same way, if someone wants to reference another note but their reference is not meant to show up along other replies to that same note, their client can choose to not include a corresponding `["e", ...]` tag for any given `nostr:nevent1...` URL inside `.content`. Clients may decide to expose these advanced functionalities to users or be more opinionated about things.

## NIP-28

## Public Chat

`draft` `optional`

This NIP defines new event kinds for public chat channels, channel messages, and basic client-side moderation.

It reserves five event kinds (40-44) for immediate use:

- `40 - channel create`
- `41 - channel metadata`
- `42 - channel message`
- `43 - hide message`
- `44 - mute user`

Client-centric moderation gives client developers discretion over what types of content they want included in their apps, while imposing no additional requirements on relays.

### Kind 40: Create channel

Create a public chat channel.

In the channel creation `content` field, Client SHOULD include basic channel metadata (`name`, `about`, `picture` and `relays` as specified in kind 41).

```
{
  "content": "{\"name\": \"Demo Channel\", \"about\": \"A test channel.\",
\"picture\": \"https://placekitten.com/200/200\", \"relays\": [\"wss://nos.lol\",
\"wss://nostr.mom\"]}",
 // other fields...
}
```

### Kind 41: Set channel metadata

Update a channel's public metadata.

Kind 41 is used to update the metadata without modifying the event id for the channel. Only the most recent kind 41 per `e` tag value MAY be available.

Clients SHOULD ignore kind 41s from pubkeys other than the kind 40 pubkey.

Clients SHOULD support basic metadata fields:

- `name` - string - Channel name
- `about` - string - Channel description
- `picture` - string - URL of channel picture
- `relays` - array - List of relays to download and broadcast events to

Clients MAY add additional metadata fields.

Clients SHOULD use NIP-10 marked "e" tags to recommend a relay.

```
{
  "content": "{\"name\": \"Updated Demo Channel\", \"about\": \"Updating a test
channel.\", \"picture\": \"https://placekitten.com/201/201\", \"relays\":
[\"wss://nos.lol\", \"wss://nostr.mom\"]}",
  "tags": [["e", <channel_create_event_id>, <relay-url>]],
  // other fields...
}
```

### Kind 42: Create channel message

Send a text message to a channel.

Clients SHOULD use NIP-10 marked "e" tags to recommend a relay and specify whether it is a reply or root message.

Clients SHOULD append NIP-10 "p" tags to replies.

Root message:

```
{
  "content": <string>,
  "tags": [["e", <kind_40_event_id>, <relay-url>, "root"]],
  // other fields...
}
```

Reply to another message:

```
{
  "content": <string>,
  "tags": [
      ["e", <kind_40_event_id>, <relay-url>, "root"],
      ["e", <kind_42_event_id>, <relay-url>, "reply"],
      ["p", <pubkey>, <relay-url>],
      // rest of tags...
  ],
  // other fields...
}
```

### Kind 43: Hide message

User no longer wants to see a certain message.

The `content` may optionally include metadata such as a `reason`.

Clients SHOULD hide event 42s shown to a given user, if there is an event 43 from that user matching the event 42 `id`.

Clients MAY hide event 42s for other users other than the user who sent the event 43.

(For example, if three users 'hide' an event giving a reason that includes the word 'pornography', a Nostr client that is an iOS app may choose to hide that message for all iOS clients.)

```
{
  "content": "{\"reason\": \"Dick pic\"}",
  "tags": [["e", <kind_42_event_id>]],
  // other fields...
}
```

### Kind 44: Mute user

User no longer wants to see messages from another user.

The `content` may optionally include metadata such as a `reason`.

Clients SHOULD hide event 42s shown to a given user, if there is an event 44 from that user matching the event 42 `pubkey`.

Clients MAY hide event 42s for users other than the user who sent the event 44.

```
{
  "content": "{\"reason\": \"Posting dick pics\"}",
  "tags": [["p", <pubkey>]],
  // other fields...
}
```

### Relay recommendations

Clients SHOULD use the relay URLs of the metadata events.

Clients MAY use any relay URL. For example, if a relay hosting the original kind 40 event for a channel goes offline, clients could instead fetch channel data from a backup relay, or a relay that clients trust more than the original relay.

## Motivation

If we're solving censorship-resistant communication for social media, we may as well solve it also for Telegram-style messaging.

We can bring the global conversation out from walled gardens into a true public square open to all.

## Additional info

- Chat demo PR with fiatjaf+jb55 comments
- Conversation about NIP16

## NIP-29

## Relay-based Groups

`draft` `optional`

This NIP defines a standard for groups that are only writable by a closed set of users. They can be public for reading by external users or not.

Groups are identified by a random string of any length that serves as an *id*.

There is no way to create a group, what happens is just that relays (most likely when asked by users) will create rules around some specific ids so these ids can serve as an actual group, henceforth messages sent to that group will be subject to these rules.

Normally a group will originally belong to one specific relay, but the community may choose to move the group to other relays or even fork the group so it exists in different forms – still using the same *id* – across different relays.

### Relay-generated events

Relays are supposed to generate the events that describe group metadata and group admins. These are *addressable* events signed by the relay keypair directly, with the group *id* as the `d` tag.

### Group identifier

A group may be identified by a string in the format `<host>'<group-id>`. For example, a group with *id* `abcdef` hosted at the relay `wss://groups.nostr.com` would be identified by the string `groups.nostr.com'abcdef`.

Group identifiers must be strings restricted to the characters `a-z0-9-_`, and SHOULD be random in order to avoid name collisions.

When encountering just the `<host>` without the `'<group-id>`, clients MAY infer `_` as the group id, which is a special top-level group dedicated to relay-local discussions.

### The `h` tag

Events sent by users to groups (chat messages, text notes, moderation events etc) MUST have an `h` tag with the value set to the group *id*.

### Timeline references

In order to not be used out of context, events sent to these groups may contain references to previous events seen from the same relay in the `previous` tag. The choice of which previous events to pick belongs to the clients. The references are to be made using the first 8 characters (4 bytes) of any event in the last 50 events seen by the user in the relay, excluding events by themselves. There can be any number of references (including zero), but it's recommended that clients include at least 3 and that relays enforce this.

This is a hack to prevent messages from being broadcasted to external relays that have forks of one group out of context. Relays are expected to reject any events that contain timeline references to events not found in their own database. Clients should also check these to keep relays honest about them.

### Late publication

Relays should prevent late publication (messages published now with a timestamp from days or even hours ago) unless they are open to receive a group forked or moved from another relay.

### Group management

Groups can have any number of users with elevated access. These users are identified by role labels which are arbitrarily defined by the relays (see also the description of `kind:39003`). What each role is capable of not defined in this NIP either, it's a relay policy that can vary. Roles can be assigned by other users (as long as they have the capability to add roles) by publishing a `kind:9000` event with that user's pubkey in a `p` tag and the roles afterwards (even if the user is already a group member a `kind:9000` can be issued and the user roles must just be updated).

The roles supported by the group as to having some special privilege assigned to them should be accessible on the event `kind:39003`, but the relay may also accept other role names, arbitrarily defined by clients, and just not do anything with them.

Users with any roles that have any privilege can be considered *admins* in a broad sense and be returned in the `kind:39001` event for a group.

### Unmanaged groups

Unmanaged groups are impromptu groups that can be used in any public relay unaware of NIP-29 specifics. They piggyback on relays' natural white/blacklists (or lack of) but aside from that are not actively managed and won't have any admins, group state or metadata events.

In `unmanaged` groups, everybody is considered to be a member.

Unmanaged groups can transition to managed groups, in that case the relay master key just has to publish moderation events setting the state of all groups and start enforcing the rules they choose to.

### Event definitions

These are the events expected to be found in NIP-29 groups.

#### Normal user-created events

Groups may accept any event kind, including chats, threads, long-form articles, calendar, livestreams, market announcements and so on. These should be as defined in their respective NIPs, with the addition of the `h` tag.

#### User-related group management events

These are events that can be sent by users to manage their situation in a group, they also require the `h` tag.

- *join request* (`kind:9021`)

Any user can send a kind `9021` event to the relay in order to request admission to the group. Relays MUST reject the request if the user has not been added to the group. The accompanying error message SHOULD explain whether the rejection is final, if the request is pending review, or if some other special handling is relevant (e.g. if payment is required). If a user is already a member, the event MUST be rejected with `duplicate:` as the error message prefix.

```
{
  "kind": 9021,
  "content": "optional reason",
  "tags": [
    ["h", "<group-id>"],
    ["code", "<optional-invite-code>"]
  ]
}
```

The optional `code` tag may be used by the relay to preauthorize acceptances in `closed` groups, together with the `kind:9009` `create-invite` moderation event.

- *leave request* (`kind:9022`)

Any user can send one of these events to the relay in order to be automatically removed from the group. The relay will automatically issue a `kind:9001` in response removing this user.

```
{
  "kind": 9022,
  "content": "optional reason",
  "tags": [
    ["h", "<group-id>"]
  ]
}
```

**Group state – or moderation**

These are events expected to be sent by the relay master key or by group admins – and relays should reject them if they don't come from an authorized admin. They also require the `h` tag.

- *moderation events* (`kinds:9000-9020`) (optional)

Clients can send these events to a relay in order to accomplish a moderation action. Relays must check if the pubkey sending the event is capable of performing the given action based on its role and the relay's internal policy (see also the description of `kind:39003`).

```
{
  "kind": 90xx,
  "content": "optional reason",
  "tags": [
    ["h", "<group-id>"],
    ["previous", /*...*/]
  ]
}
```

Each moderation action uses a different kind and requires different arguments, which are given as tags. These are defined in the following table:

| kind | name | tags |
|---|---|---|
| 9000 | `put-user` | `p` with pubkey hex and optional roles |
| 9001 | `remove-user` | `p` with pubkey hex |
| 9002 | `edit-metadata` | fields from `kind:39000` to be modified |
| 9005 | `delete-event` | `e` with event id hex |
| 9007 | `create-group` | |
| 9008 | `delete-group` | |
| 9009 | `create-invite` | |

It's expected that the group state (of who is an allowed member or not, who is an admin and with which permission or not, what are the group name and picture etc) can be fully reconstructed from the canonical sequence of these events.

**Group metadata events**

These events contain the group id in a `d` tag instead of the `h` tag. They MUST be created by the relay master key only and a single instance of each (or none) should exist at all times for each group. They are merely informative but should reflect the latest group state (as it was changed by moderation events over time).

- *group metadata* (`kind:39000`) (optional)

This event defines the metadata for the group – basically how clients should display it. It must be generated and signed by the relay in which is found. Relays shouldn't accept these events if they're signed by anyone else.

If the group is forked and hosted in multiple relays, there will be multiple versions of this event in each different relay and so on.

When this event is not found, clients may still connect to the group, but treat it as having a different status, `unmanaged`,

```
{
  "kind": 39000,
  "content": "",
  "tags": [
    ["d", "<group-id>"],
    ["name", "Pizza Lovers"],
    ["picture", "https://pizza.com/pizza.png"],
    ["about", "a group for people who love pizza"],
    ["public"], // or ["private"]
    ["open"] // or ["closed"]
  ]
  // other fields...
}
```

`name`, `picture` and `about` are basic metadata for the group for display purposes. `public` signals the group can be *read* by anyone, while `private` signals that only AUTHed users can read. `open` signals that anyone can request to join and the request will be automatically granted, while `closed` signals that members must be pre-approved or that requests to join will be manually handled.

- *group admins* (`kind:39001`) (optional)

Each admin is listed along with one or more roles. These roles SHOULD have a correspondence with the roles supported by the relay, as advertised by the `kind:39003` event.

```
{
  "kind": 39001,
  "content": "list of admins for the pizza lovers group",
  "tags": [
    ["d", "<group-id>"],
    ["p", "<pubkey1-as-hex>", "ceo"],
    ["p", "<pubkey2-as-hex>", "secretary", "gardener"],
    // other pubkeys...
  ],
  // other fields...
}
```

- *group members* (`kind:39002`) (optional)

It's a list of pubkeys that are members of the group. Relays might choose to not to publish this information, to restrict what pubkeys can fetch it or to only display a subset of the members in it.

Clients should not assume this will always be present or that it will contain a full list of members.

```
{
  "kind": 39002,
  "content": "list of members for the pizza lovers group",
  "tags": [
    ["d", "<group-id>"],
    ["p", "<admin1>"],
    ["p", "<member-pubkey1>"],
    ["p", "<member-pubkey2>"],
    // other pubkeys...
```

```
  ],
  // other fields...
}
```

- *group roles* (`kind:39003`) (optional)

This is an event that MAY be published by the relay informing users and clients about what are the roles supported by this relay according to its internal logic.

For example, a relay may choose to support the roles `"admin"` and `"moderator"`, in which the `"admin"` will be allowed to edit the group metadata, delete messages and remove users from the group, while the `"moderator"` can only delete messages (or the relay may choose to call these roles `"ceo"` and `"secretary"` instead, the exact role name is not relevant).

The process through which the relay decides what roles to support and how to handle moderation events internally based on them is specific to each relay and not specified here.

```
{
  "kind": 39003,
  "content": "list of roles supported by this group",
  "tags": [
    ["d", "<group-id>"],
    ["role", "<role-name>", "<optional-description>"],
    ["role", "<role-name>", "<optional-description>"],
    // other roles...
  ],
  // other fields...
}
```

### Implementation quirks

#### Checking your own membership in a group

The latest of either `kind:9000` or `kind:9001` events present in a group should tell a user that they are currently members of the group or if they were removed. In case none of these exist the user is assumed to not be a member of the group – unless the group is `unmanaged`, in which case the user is assumed to be a member.

#### Adding yourself to a group

When a group is `open`, anyone can send a `kind:9021` event to it in order to be added, then expect a `kind:9000` event to be emitted confirming that the user was added. The same happens with `closed` groups, except in that case a user may only send a `kind:9021` if it has an invite code.

#### Storing your list of groups

A definition for `kind:10009` was included in NIP-51 that allows clients to store the list of groups a user wants to remember being in.

#### Using `unmanaged` relays

To prevent event leakage, when using `unmanaged` relays, clients should include the NIP-70 `-` tag, as just the `previous` tag won't be checked by other `unmanaged` relays.

Groups MAY be named without relay support by adding a `name` to the corresponding tag in a user's `kind 10009` group list.

## NIP-30

## Custom Emoji

`draft` `optional`

Custom emoji may be added to **kind 0**, **kind 1**, **kind 7** (NIP-25 ) and **kind 30315** (NIP-38 ) events by including one or more `"emoji"` tags, in the form:

```
["emoji", <shortcode>, <image-url>]
```

Where:

- `<shortcode>` is a name given for the emoji, which MUST be comprised of only alphanumeric characters and underscores.
- `<image-url>` is a URL to the corresponding image file of the emoji.

For each emoji tag, clients should parse emoji shortcodes (aka “emojify”) like `:shortcode:` in the event to display custom emoji.

Clients may allow users to add custom emoji to an event by including `:shortcode:` identifier in the event, and adding the relevant `"emoji"` tags.

#### Kind 0 events

In kind 0 events, the `name` and `about` fields should be emojified.

```
{
  "kind": 0,
  "content": "{\"name\":\"Alex Gleason :soapbox:\"}",
  "tags": [
    ["emoji", "soapbox", "https://gleasonator.com/emoji/Gleasonator/soapbox.png"]
  ],
  "pubkey": "79c2cae114ea28a981e7559b4fe7854a473521a8d22a66bbab9fa248eb820ff6",
  "created_at": 1682790000
}
```

#### Kind 1 events

In kind 1 events, the `content` should be emojified.

```
{
  "kind": 1,
  "content": "Hello :gleasonator: 😂 :ablobcatrainbow: :disputed: yolo",
    "tags": [
    ["emoji", "ablobcatrainbow",
"https://gleasonator.com/emoji/blobcat/ablobcatrainbow.png"],
    ["emoji", "disputed", "https://gleasonator.com/emoji/Fun/disputed.png"],
    ["emoji", "gleasonator",
"https://gleasonator.com/emoji/Gleasonator/gleasonator.png"]
  ],
  "pubkey": "79c2cae114ea28a981e7559b4fe7854a473521a8d22a66bbab9fa248eb820ff6",
  "created_at": 1682630000
}
```

#### Kind 7 events

In kind 7 events, the `content` should be emojified.

```
{
  "kind": 7,
  "content": ":dezh:",
  "tags": [
    ["emoji", "dezh", "https://raw.githubusercontent.com/dezh-tech/brand-
assets/main/dezh/logo/black-normal.svg"]
  ],
  "pubkey": "79c2cae114ea28a981e7559b4fe7854a473521a8d22a66bbab9fa248eb820ff6",
  "created_at": 1682630000
}
```

## NIP-31

## Dealing with unknown event kinds

`draft` `optional`

When creating a new custom event kind that is part of a custom protocol and isn’t meant to be read as text (like `kind:1`), clients should use an `alt` tag to write a short human-readable plaintext summary of what that event is about.

The intent is that social clients, used to display only `kind:1` notes, can still show something in case a custom event pops up in their timelines. The content of the `alt` tag should provide enough context for a user that doesn't know anything about this event kind to understand what it is.

These clients that only know `kind:1` are not expected to ask relays for events of different kinds, but users could still reference these weird events on their notes, and without proper context these could be nonsensical notes. Having the fallback text makes that situation much better – even if only for making the user aware that they should try to view that custom event elsewhere.

`kind:1`-centric clients can make interacting with these event kinds more functional by supporting NIP-89 .

## NIP-32

## Labeling

`draft` `optional`

This NIP defines two new indexable tags to label events and a new event kind (`kind:1985`) to attach those labels to existing events. This supports several use cases, including distributed moderation, collection management, license assignment, and content classification.

New Tags:

- `L` denotes a label namespace
- `l` denotes a label

## Label Namespace Tag

An `L` tag can be any string, but publishers SHOULD ensure they are unambiguous by using a well-defined namespace (such as an ISO standard) or reverse domain name notation.

`L` tags are RECOMMENDED in order to support searching by namespace rather than by a specific tag. The special `ugc` ("user generated content") namespace MAY be used when the label content is provided by an end user.

`L` tags starting with `#` indicate that the label target should be associated with the label's value. This is a way of attaching standard nostr tags to events, pubkeys, relays, urls, etc.

## Label Tag

An `l` tag's value can be any string. If using an `L` tag, `l` tags MUST include a mark matching an `L` tag value in the same event. If no `L` tag is included, a mark SHOULD still be included. If none is included, `ugc` is implied.

## Label Target

The label event MUST include one or more tags representing the object or objects being labeled: `e`, `p`, `a`, `r`, or `t` tags. This allows for labeling of events, people, relays, or topics respectively. As with NIP-01, a relay hint SHOULD be included when using `e` and `p` tags.

## Content

Labels should be short, meaningful strings. Longer discussions, such as for an explanation of why something was labeled the way it was, should go in the event's `content` field.

## Self-Reporting

`l` and `L` tags MAY be added to other event kinds to support self-reporting. For events with a kind other than 1985, labels refer to the event itself.

## Example events

A suggestion that multiple pubkeys be associated with the `permies` topic.

```
{
  "kind": 1985,
  "tags": [
    ["L", "#t"],
    ["l", "permies", "#t"],
    ["p", <pubkey1>, <relay_url>],
    ["p", <pubkey2>, <relay_url>]
```

```
  ],
  // other fields...
}
```

A report flagging violence toward a human being as defined by ontology.example.com.

```
{
  "kind": 1985,
  "tags": [
    ["L", "com.example.ontology"],
    ["l", "VI-hum", "com.example.ontology"],
    ["p", <pubkey1>, <relay_url>],
    ["p", <pubkey2>, <relay_url>]
  ],
  // other fields...
}
```

A moderation suggestion for a chat event.

```
{
  "kind": 1985,
  "tags": [
    ["L", "nip28.moderation"],
    ["l", "approve", "nip28.moderation"],
    ["e", <kind40_event_id>, <relay_url>]
  ],
  // other fields...
}
```

Assignment of a license to an event.

```
{
  "kind": 1985,
  "tags": [
    ["L", "license"],
    ["l", "MIT", "license"],
    ["e", <event_id>, <relay_url>]
  ],
  // other fields...
}
```

Publishers can self-label by adding `l` tags to their own non-1985 events. In this case, the kind 1 event's author is labeling their note as being related to Milan, Italy using ISO 3166-2.

```
{
  "kind": 1,
  "tags": [
    ["L", "ISO-3166-2"],
    ["l", "IT-MI", "ISO-3166-2"]
  ],
  "content": "It's beautiful here in Milan!",
  // other fields...
}
```

Author is labeling their note language as English using ISO-639-1.

```
{
  "kind": 1,
  "tags": [
    ["L", "ISO-639-1"],
    ["l", "en", "ISO-639-1"]
  ],
  "content": "English text",
  // other fields...
}
```

## Other Notes

When using this NIP to bulk-label many targets at once, events may be requested for deletion using NIP-09 and a replacement may be published. We have opted not to use addressable/replaceable events for this due to the complexity in coming up with a standard `d` tag. In order to avoid ambiguity when querying, publishers SHOULD limit labeling events to a single namespace.

Before creating a vocabulary, explore how your use case may have already been designed and imitate that design if possible. Reverse domain name notation is encouraged to avoid namespace clashes, but for the sake of interoperability all namespaces should be considered open for public use, and not proprietary. In other words, if there is a namespace that fits your use case, use it even if it points to someone else's domain name.

Vocabularies MAY choose to fully qualify all labels within a namespace (for example, `["l", "com.example.vocabulary:my-label"]`). This may be preferred when defining more formal vocabularies that should not be confused with another namespace when querying without an `L` tag. For these vocabularies, all labels SHOULD include the namespace (rather than mixing qualified and unqualified labels).

A good heuristic for whether a use case fits this NIP is whether labels would ever be unique. For example, many events might be labeled with a particular place, topic, or pubkey, but labels with specific values like "John Doe" or "3.18743" are not labels, they are values, and should be handled in some other way.

## Appendix: Known Ontologies

Below is a non-exhaustive list of ontologies currently in widespread use.

- social ontology categories

## NIP-33

## Parameterized Replaceable Events

`final` `mandatory`

Renamed to "Addressable events" and moved to NIP-01 .

## NIP-34

## `git` stuff

`draft` `optional`

This NIP defines all the ways code collaboration using and adjacent to `git` can be done using Nostr.

### Repository announcements

Git repositories are hosted in Git-enabled servers, but their existence can be announced using Nostr events, as well as their willingness to receive patches, bug reports and comments in general.

```
{
  "kind": 30617,
  "content": "",
  "tags": [
    ["d", "<repo-id>"], // usually kebab-case short name
    ["name", "<human-readable project name>"],
    ["description", "brief human-readable project description>"],
    ["web", "<url for browsing>", ...], // a webpage url, if the git server being
used provides such a thing
    ["clone", "<url for git-cloning>", ...], // a url to be given to `git clone` so
anyone can clone it
    ["relays", "<relay-url>", ...] // relays that this repository will monitor for
patches and issues
    ["r", "<earliest-unique-commit-id>", "euc"]
    ["maintainers", "<other-recognized-maintainer>", ...]
  ]
}
```

The tags `web`, `clone`, `relays`, `maintainers` can have multiple values.

The `r` tag annotated with the `"euc"` marker should be the commit ID of the earliest unique commit of this repo, made to identify it among forks and group it with other repositories hosted elsewhere that may represent essentially the same project. In most cases it will be the root commit of a repository. In case of a permanent fork between two projects, then the first commit after the fork should be used.

Except `d`, all tags are optional.

### Repository state announcements

An optional source of truth for the state of branches and tags in a repository.

```
{
  "kind": 30618,
  "content": "",
  "tags": [
    ["d", "<repo-id>"], // matches the identifier in the coresponding repository
announcement
    ["refs/<heads|tags>/<branch-or-tag-name>","<commit-id>"]
    ["HEAD", "ref: refs/heads/<branch-name>"]
  ]
}
```

The `refs` tag may appear multiple times, or none.

If no `refs` tags are present, the author is no longer tracking repository state using this event. This approach enables the author to restart tracking state at a later time unlike NIP-09 deletion requests.

The `refs` tag can be optionally extended to enable clients to identify how many commits ahead a ref is:

```
{
  "tags": [
    ["refs/<heads|tags>/<branch-or-tag-name>", "<commit-id>", "<shorthand-parent-
commit-id>", "<shorthand-grandparent>", ...],
  ]
}
```

### Patches

Patches can be sent by anyone to any repository. Patches to a specific repository SHOULD be sent to the relays specified in that repository's announcement event's `"relays"` tag. Patch events SHOULD include an `a` tag pointing to that repository's announcement address.

Patches in a patch set SHOULD include a NIP-10 `e reply` tag pointing to the previous patch.

The first patch revision in a patch revision SHOULD include a NIP-10 `e reply` to the original root patch.

```
{
  "kind": 1617,
  "content": "<patch>", // contents of <git format-patch>
  "tags": [
    ["a", "30617:<base-repo-owner-pubkey>:<base-repo-id>"],
    ["r", "<earliest-unique-commit-id-of-repo>"] // so clients can subscribe to all
patches sent to a local git repo
    ["p", "<repository-owner>"],
    ["p", "<other-user>"], // optionally send the patch to another user to bring it
to their attention

    ["t", "root"], // omitted for additional patches in a series
    // for the first patch in a revision
    ["t", "root-revision"],

    // optional tags for when it is desirable that the merged patch has a stable
commit id
    // these fields are necessary for ensuring that the commit resulting from
applying a patch
    // has the same id as it had in the proposer's machine -- all these tags can be
omitted
    // if the maintainer doesn't care about these things
    ["commit", "<current-commit-id>"],
    ["r", "<current-commit-id>"] // so clients can find existing patches for a
specific commit
    ["parent-commit", "<parent-commit-id>"],
    ["commit-pgp-sig", "-----BEGIN PGP SIGNATURE-----..."], // empty string for
unsigned commit
```

```
    ["committer", "<name>", "<email>", "<timestamp>", "<timezone offset in
minutes>"],
  ]
}
```

The first patch in a series MAY be a cover letter in the format produced by `git format-patch`.

### Issues

Issues are Markdown text that is just human-readable conversational threads related to the repository: bug reports, feature requests, questions or comments of any kind. Like patches, these SHOULD be sent to the relays specified in that repository's announcement event's `"relays"` tag.

Issues may have a `subject` tag, which clients can utilize to display a header. Additionally, one or more `t` tags may be included to provide labels for the issue.

```
{
  "kind": 1621,
  "content": "<markdown text>",
  "tags": [
    ["a", "30617:<base-repo-owner-pubkey>:<base-repo-id>"],
    ["p", "<repository-owner>"]
    ["subject", "<issue-subject>"]
    ["t", "<issue-label>"]
    ["t", "<another-issue-label>"]
  ]
}
```

### Replies

Replies are also Markdown text. The difference is that they MUST be issued as replies to either a `kind:1621` *issue* or a `kind:1617` *patch* event. The threading of replies and patches should follow NIP-10 rules.

```
{
  "kind": 1622,
  "content": "<markdown text>",
  "tags": [
    ["a", "30617:<base-repo-owner-pubkey>:<base-repo-id>", "<relay-url>"],
    ["e", "<issue-or-patch-id-hex>", "", "root"],

    // other "e" and "p" tags should be applied here when necessary, following the
threading rules of NIP-10
    ["p", "<patch-author-pubkey-hex>", "", "mention"],
    ["e", "<previous-reply-id-hex>", "", "reply"],
    // rest of tags...
  ],
  // other fields...
}
```

### Status

Root Patches and Issues have a Status that defaults to 'Open' and can be set by issuing Status events.

```
{
  "kind": 1630, // Open
  "kind": 1631, // Applied / Merged for Patches; Resolved for Issues
  "kind": 1632, // Closed
  "kind": 1633, // Draft
  "content": "<markdown text>",
  "tags": [
    ["e", "<issue-or-original-root-patch-id-hex>", "", "root"],
    ["e", "<accepted-revision-root-id-hex>", "", "reply"], // for when revisions
applied
    ["p", "<repository-owner>"],
    ["p", "<root-event-author>"],
    ["p", "<revision-author>"],

    // optional for improved subscription filter efficiency
```

```
    ["a", "30617:<base-repo-owner-pubkey>:<base-repo-id>", "<relay-url>"],
    ["r", "<earliest-unique-commit-id-of-repo>"]

    // optional for `1631` status
    ["e", "<applied-or-merged-patch-event-id>", "", "mention"], // for each
    // when merged
    ["merge-commit", "<merge-commit-id>"]
    ["r", "<merge-commit-id>"]
    // when applied
    ["applied-as-commits", "<commit-id-in-master-branch>", ...]
    ["r", "<applied-commit-id>"] // for each
  ]
}
```

The Status event with the largest created_at date is valid.

The Status of a patch-revision defaults to either that of the root-patch, or `1632` (Closed) if the root-patch's Status is `1631` and the patch-revision isn't tagged in the `1631` event.

### Possible things to be added later

- "branch merge" kind (specifying a URL from where to fetch the branch to be merged)
- inline file comments kind (we probably need one for patches and a different one for merged files)

## NIP-35

## Torrents

`draft` `optional`

This NIP defined a new `kind 2003` which is a Torrent.

`kind 2003` is a simple torrent index where there is enough information to search for content and construct the magnet link. No torrent files exist on nostr.

### Tags

- `x`: V1 BitTorrent Info Hash, as seen in the magnet link `magnet:?xt=urn:btih:HASH`
- `file`: A file entry inside the torrent, including the full path ie. `info/example.txt`
- `tracker`: (Optional) A tracker to use for this torrent

In order to make torrents searchable by general category, you SHOULD include a few tags like `movie`, `tv`, `HD`, `UHD` etc.

### Tag prefixes

Tag prefixes are used to label the content with references, ie. `["i", "imdb:1234"]`

- `tcat`: A comma separated text category path, ie. `["i", "tcat:video,movie,4k"]`, this should also match the `newznab` category in a best effort approach.
- `newznab`: The category ID from newznab
- `tmdb`: The movie database id.
- `ttvdb`: TV database id.
- `imdb`: IMDB id.
- `mal`: MyAnimeList id.
- `anilist`: AniList id.

A second level prefix should be included where the database supports multiple media types.

- `tmdb:movie:693134` maps to `themoviedb.org/movie/693134`
- `ttvdb:movie:290272` maps to `thetvdb.com/movies/dune-part-two`
- `mal:anime:9253` maps to `myanimelist.net/anime/9253`
- `mal:manga:17517` maps to `myanimelist.net/manga/17517`

In some cases the url mapping isnt direct, mapping the url in general is out of scope for this NIP, the section above is only a guide so that implementers have enough information to succsesfully map the url if they wish.

```
{
  "kind": 2003,
  "content": "<long-description-pre-formatted>",
```

```
  "tags": [
    ["title", "<torrent-title>"],
    ["x", "<bittorrent-info-hash>"],
    ["file", "<file-name>", "<file-size-in-bytes>"],
    ["file", "<file-name>", "<file-size-in-bytes>"],
    ["tracker", "udp://mytacker.com:1337"],
    ["tracker", "http://1337-tracker.net/announce"],
    ["i", "tcat:video,movie,4k"],
    ["i", "newznab:2045"],
    ["i", "imdb:tt15239678"],
    ["i", "tmdb:movie:693134"],
    ["i", "ttvdb:movie:290272"],
    ["t", "movie"],
    ["t", "4k"],
  ]
}
```

### Torrent Comments

A torrent comment is a `kind 2004` event which is used to reply to a torrent event.

This event works exactly like a `kind 1` and should follow `NIP-10` for tagging.

### Implementations

1. dtan.xyz
2. nostrudel.ninja NIP-36 ======

## Sensitive Content / Content Warning

`draft` `optional`

The `content-warning` tag enables users to specify if the event's content needs to be approved by readers to be shown. Clients can hide the content until the user acts on it.

`l` and `L` tags MAY be also be used as defined in NIP-32 with the `content-warning` or other namespace to support further qualification and querying.

#### Spec

```
tag: content-warning
options:
 - [reason]: optional
```

#### Example

```
{
  "pubkey": "<pub-key>",
  "created_at": 1000000000,
  "kind": 1,
  "tags": [
    ["t", "hastag"],
    ["L", "content-warning"],
    ["l", "reason", "content-warning"],
    ["L", "social.nos.ontology"],
    ["l", "NS-nud", "social.nos.ontology"],
    ["content-warning", "<optional reason>"]
  ],
  "content": "sensitive content with #hastag\n",
  "id": "<event-id>"
}
```

## NIP-37

## Draft Events

`draft` `optional`

This NIP defines kind `31234` as a private wrap for drafts of any other event kind.

The draft event is JSON-stringified, NIP44-encrypted to the signer's public key and placed inside the `.content` of the event.

An additional `k` tag identifies the kind of the draft event.

```
{
  "kind": 31234,
  "tags": [
    ["d", "<identifier>"],
    ["k", "<kind of the draft event>"],
    ["e", "<anchor event event id>", "<relay-url>"],
    ["a", "<anchor event address>", "<relay-url>"],
  ],
  "content": nip44Encrypt(JSON.stringify(draft_event)),
  // other fields
}
```

A blanked `.content` means this draft has been deleted by a client but relays still have the event.

Tags `e` and `a` identify one or more anchor events, such as parent events on replies.

### Relay List for Private Content

Kind `10013` indicates the user's preferred relays to store private events like Drafts. The event MUST include a list of `relay` URLs in private tags. Private tags are JSON Stringified, NIP-44-encrypted to the signer's keys and placed inside the .content of the event.

```
{
  "kind": 10013,
  "tags": [],
  "content": nip44Encrypt(JSON.stringify([
    ["relay", "wss://myrelay.mydomain.com"]
  ]))
  //...other fields
}
```

Relays listed in this event SHOULD be authed and only allow downloads to events signed by the authed user.

Clients SHOULD publish kind `10013` events to the author's NIP-65 `write` relays.

## NIP-38

## User Statuses

`draft` `optional`

### Abstract

This NIP enables a way for users to share live statuses such as what music they are listening to, as well as what they are currently doing: work, play, out of office, etc.

### Live Statuses

A special event with `kind:30315` "User Status" is defined as an *optionally expiring addressable event*, where the `d` tag represents the status type:

For example:

```
{
  "kind": 30315,
  "content": "Sign up for nostrasia!",
  "tags": [
    ["d", "general"],
    ["r", "https://nostr.world"]
  ],
}
```

```
{
  "kind": 30315,
  "content": "Intergalatic - Beastie Boys",
  "tags": [
    ["d", "music"],
    ["r", "spotify:search:Intergalatic%20-%20Beastie%20Boys"],
    ["expiration", "1692845589"]
  ],
}
```

Two common status types are defined: `general` and `music`. `general` represent general statuses: “Working”, “Hiking”, etc.

`music` status events are for live streaming what you are currently listening to. The expiry of the `music` status should be when the track will stop playing.

Any other status types can be used but they are not defined by this NIP.

The status MAY include an `r`, `p`, `e` or `a` tag linking to a URL, profile, note, or addressable event.

The `content` MAY include emoji(s), or NIP-30 custom emoji(s). If the `content` is an empty string then the client should clear the status.

## Client behavior

Clients MAY display this next to the username on posts or profiles to provide live user status information.

## Use Cases

- Calendar nostr apps that update your general status when you’re in a meeting
- Nostr Nests that update your general status with a link to the nest when you join
- Nostr music streaming services that update your music status when you’re listening
- Podcasting apps that update your music status when you’re listening to a podcast, with a link for others to listen as well
- Clients can use the system media player to update playing music status

## NIP-39

## External Identities in Profiles

`draft` `optional`

### Abstract

Nostr protocol users may have other online identities such as usernames, profile pages, keypairs etc. they control and they may want to include this data in their profile metadata so clients can parse, validate and display this information.

### `i` tag on a metadata event

A new optional `i` tag is introduced for `kind 0` metadata event defined in NIP-01 :

```
{
  "id": <id>,
  "pubkey": <pubkey>,
  "tags": [
    ["i", "github:semisol", "9721ce4ee4fceb91c9711ca2a6c9a5ab"],
    ["i", "twitter:semisol_public", "1619358434134196225"],
    ["i", "mastodon:bitcoinhackers.org/@semisol", "109775066355589974"]
    ["i", "telegram:1087295469", "nostrdirectory/770"]
  ],
  // other fields...
}
```

An `i` tag will have two parameters, which are defined as the following:

1. `platform:identity`: This is the platform name (for example `github`) and the identity on that platform (for example `semisol`) joined together with `:`.
2. `proof`: String or object that points to the proof of owning this identity.

Clients SHOULD process any `i` tags with more than 2 values for future extensibility. Identity provider names SHOULD only include `a-z`, `0-9` and the characters `._-/` and MUST NOT include `:`. Identity names SHOULD be normalized if possible by replacing uppercase letters with lowercase letters, and if there are multiple aliases for an entity the primary one should be used.

### Claim types

#### `github`

Identity: A GitHub username.

Proof: A GitHub Gist ID. This Gist should be created by `<identity>` with a single file that has the text `Verifying that I control the following Nostr public key: <npub encoded public key>`. This can be located at `https://gist.github.com/<identity>/<proof>`.

#### `twitter`

Identity: A Twitter username.

Proof: A Tweet ID. The tweet should be posted by `<identity>` and have the text `Verifying my account on nostr My Public Key: "<npub encoded public key>"`. This can be located at `https://twitter.com/<identity>/status/<proof>`.

#### `mastodon`

Identity: A Mastodon instance and username in the format `<instance>/@<username>`.

Proof: A Mastodon post ID. This post should be published by `<username>@<instance>` and have the text `Verifying that I control the following Nostr public key: "<npub encoded public key>"`. This can be located at `https://<identity>/<proof>`.

#### `telegram`

Identity: A Telegram user ID.

Proof: A string in the format `<ref>/<id>` which points to a message published in the public channel or group with name `<ref>` and message ID `<id>`. This message should be sent by user ID `<identity>` and have the text `Verifying that I control the following Nostr public key: "<npub encoded public key>"`. This can be located at `https://t.me/<proof>`.

## NIP-40

## Expiration Timestamp

`draft` `optional`

The `expiration` tag enables users to specify a unix timestamp at which the message SHOULD be considered expired (by relays and clients) and SHOULD be deleted by relays.

#### Spec

```
tag: expiration
values:
 - [UNIX timestamp in seconds]: required
```

#### Example

```
{
  "pubkey": "<pub-key>",
  "created_at": 1000000000,
  "kind": 1,
  "tags": [
    ["expiration", "1600000000"]
  ],
  "content": "This message will expire at the specified timestamp and be deleted by
relays.\n",
  "id": "<event-id>"
}
```

Note: The timestamp should be in the same format as the created_at timestamp and should be interpreted as the time at which the message should be deleted by relays.

## Client Behavior

Clients SHOULD use the `supported_nips` field to learn if a relay supports this NIP. Clients SHOULD NOT send expiration events to relays that do not support this NIP.

Clients SHOULD ignore events that have expired.

## Relay Behavior

Relays MAY NOT delete expired messages immediately on expiration and MAY persist them indefinitely. Relays SHOULD NOT send expired events to clients, even if they are stored. Relays SHOULD drop any events that are published to them if they are expired. An expiration timestamp does not affect storage of ephemeral events.

## Suggested Use Cases

- Temporary announcements - This tag can be used to make temporary announcements. For example, an event organizer could use this tag to post announcements about an upcoming event.
- Limited-time offers - This tag can be used by businesses to make limited-time offers that expire after a certain amount of time. For example, a business could use this tag to make a special offer that is only available for a limited time.

#### Warning

The events could be downloaded by third parties as they are publicly accessible all the time on the relays. So don't consider expiring messages as a security feature for your conversations or other uses.

## NIP-42

## Authentication of clients to relays

`draft` `optional`

This NIP defines a way for clients to authenticate to relays by signing an ephemeral event.

### Motivation

A relay may want to require clients to authenticate to access restricted resources. For example,

- A relay may request payment or other forms of whitelisting to publish events – this can naïvely be achieved by limiting publication to events signed by the whitelisted key, but with this NIP they may choose to accept any events as long as they are published from an authenticated user;
- A relay may limit access to `kind: 4` DMs to only the parties involved in the chat exchange, and for that it may require authentication before clients can query for that kind.
- A relay may limit subscriptions of any kind to paying users or users whitelisted through any other means, and require authentication.

### Definitions

#### New client-relay protocol messages

This NIP defines a new message, `AUTH`, which relays CAN send when they support authentication and clients can send to relays when they want to authenticate. When sent by relays the message has the following form:

```
["AUTH", <challenge-string>]
```

And, when sent by clients, the following form:

```
["AUTH", <signed-event-json>]
```

`AUTH` messages sent by clients MUST be answered with an `OK` message, like any `EVENT` message.

#### Canonical authentication event

The signed event is an ephemeral event not meant to be published or queried, it must be of `kind: 22242` and it should have at least two tags, one for the relay URL and one for the challenge string as received from the relay. Relays MUST exclude `kind: 22242` events from being broadcasted to any client. `created_at` should be the current time. Example:

```
{
  "kind": 22242,
  "tags": [
    ["relay", "wss://relay.example.com/"],
    ["challenge", "challengestringhere"]
  ],
  // other fields...
}
```

#### `OK` and `CLOSED` machine-readable prefixes

This NIP defines two new prefixes that can be used in `OK` (in response to event writes by clients) and `CLOSED` (in response to rejected subscriptions by clients):

- `"auth-required: "` - for when a client has not performed `AUTH` and the relay requires that to fulfill the query or write the event.
- `"restricted: "` - for when a client has already performed `AUTH` but the key used to perform it is still not allowed by the relay or is exceeding its authorization.

### Protocol flow

At any moment the relay may send an `AUTH` message to the client containing a challenge. The challenge is valid for the duration of the connection or until another challenge is sent by the relay. The client MAY decide to send its `AUTH` event at any point and the authenticated session is valid afterwards for the duration of the connection.

#### `auth-required` in response to a `REQ` message

Given that a relay is likely to require clients to perform authentication only for certain jobs, like answering a `REQ` or accepting an `EVENT` write, these are some expected common flows:

```
relay: ["AUTH", "<challenge>"]
client: ["REQ", "sub_1", {"kinds": [4]}]
relay: ["CLOSED", "sub_1", "auth-required: we can't serve DMs to unauthenticated
users"]
client: ["AUTH", {"id": "abcdef...", ...}]
relay: ["OK", "abcdef...", true, ""]
client: ["REQ", "sub_1", {"kinds": [4]}]
relay: ["EVENT", "sub_1", {...}]
relay: ["EVENT", "sub_1", {...}]
relay: ["EVENT", "sub_1", {...}]
relay: ["EVENT", "sub_1", {...}]
...
```

In this case, the `AUTH` message from the relay could be sent right as the client connects or it can be sent immediately before the `CLOSED` is sent. The only requirement is that *the client must have a stored challenge associated with that relay* so it can act upon that in response to the `auth-required` `CLOSED` message.

#### `auth-required` in response to an `EVENT` message

The same flow is valid for when a client wants to write an `EVENT` to the relay, except now the relay sends back an `OK` message instead of a `CLOSED` message:

```
relay: ["AUTH", "<challenge>"]
client: ["EVENT", {"id": "012345...", ...}]
relay: ["OK", "012345...", false, "auth-required: we only accept events from
registered users"]
client: ["AUTH", {"id": "abcdef...", ...}]
relay: ["OK", "abcdef...", true, ""]
client: ["EVENT", {"id": "012345...", ...}]
relay: ["OK", "012345...", true, ""]
```

### Signed Event Verification

To verify `AUTH` messages, relays must ensure:

- that the `kind` is `22242`;
- that the event `created_at` is close (e.g. within ~10 minutes) of the current time;

- that the "`challenge`" tag matches the challenge sent before;
- that the "`relay`" tag matches the relay URL:
    - URL normalization techniques can be applied. For most cases just checking if the domain name is correct should be enough.

## NIP-44

## Encrypted Payloads (Versioned)

`optional`

The NIP introduces a new data format for keypair-based encryption. This NIP is versioned to allow multiple algorithm choices to exist simultaneously. This format may be used for many things, but MUST be used in the context of a signed event as described in NIP-01.

*Note*: this format DOES NOT define any `kind`s related to a new direct messaging standard, only the encryption required to define one. It SHOULD NOT be used as a drop-in replacement for NIP-04 payloads.

### Versions

Currently defined encryption algorithms:

- `0x00` - Reserved
- `0x01` - Deprecated and undefined
- `0x02` - secp256k1 ECDH, HKDF, padding, ChaCha20, HMAC-SHA256, base64

### Limitations

Every nostr user has their own public key, which solves key distribution problems present in other solutions. However, nostr's relay-based architecture makes it difficult to implement more robust private messaging protocols with things like metadata hiding, forward secrecy, and post compromise secrecy.

The goal of this NIP is to have a *simple* way to encrypt payloads used in the context of a signed event. When applying this NIP to any use case, it's important to keep in mind your users' threat model and this NIP's limitations. For high-risk situations, users should chat in specialized E2EE messaging software and limit use of nostr to exchanging contacts.

On its own, messages sent using this scheme have a number of important shortcomings:

- No deniability: it is possible to prove an event was signed by a particular key
- No forward secrecy: when a key is compromised, it is possible to decrypt all previous conversations
- No post-compromise security: when a key is compromised, it is possible to decrypt all future conversations
- No post-quantum security: a powerful quantum computer would be able to decrypt the messages
- IP address leak: user IP may be seen by relays and all intermediaries between user and relay
- Date leak: `created_at` is public, since it is a part of NIP-01 event
- Limited message size leak: padding only partially obscures true message length
- No attachments: they are not supported

Lack of forward secrecy may be partially mitigated by only sending messages to trusted relays, and asking relays to delete stored messages after a certain duration has elapsed.

### Version 2

NIP-44 version 2 has the following design characteristics:

- Payloads are authenticated using a MAC before signing rather than afterwards because events are assumed to be signed as specified in NIP-01. The outer signature serves to authenticate the full payload, and MUST be validated before decrypting.
- ChaCha is used instead of AES because it's faster and has better security against multi-key attacks .
- ChaCha is used instead of XChaCha because XChaCha has not been standardized. Also, xChaCha's improved collision resistance of nonces isn't necessary since every message has a new (key, nonce) pair.
- HMAC-SHA256 is used instead of Poly1305 because polynomial MACs are much easier to forge.
- SHA256 is used instead of SHA3 or BLAKE because it is already used in nostr. Also BLAKE's speed advantage is smaller in non-parallel environments.
- A custom padding scheme is used instead of padmé because it provides better leakage reduction for small messages.
- Base64 encoding is used instead of another encoding algorithm because it is widely available, and is already used in nostr.

#### Encryption

1. Calculate a conversation key
    - Execute ECDH (scalar multiplication) of public key B by private key A Output `shared_x` must be unhashed, 32-byte encoded x coordinate of the shared point
    - Use HKDF-extract with sha256, `IKM=shared_x` and `salt=utf8_encode('nip44-v2')`
    - HKDF output will be a `conversation_key` between two users.
    - It is always the same, when key roles are swapped: `conv(a, B) == conv(b, A)`
2. Generate a random 32-byte nonce
    - Always use CSPRNG
    - Don't generate a nonce from message content
    - Don't re-use the same nonce between messages: doing so would make them decryptable, but won't leak the long-term key
3. Calculate message keys
    - The keys are generated from `conversation_key` and `nonce`. Validate that both are 32 bytes long
    - Use HKDF-expand, with sha256, `PRK=conversation_key`, `info=nonce` and `L=76`
    - Slice 76-byte HKDF output into: `chacha_key` (bytes 0..32), `chacha_nonce` (bytes 32..44), `hmac_key` (bytes 44..76)
4. Add padding
    - Content must be encoded from UTF-8 into byte array
    - Validate plaintext length. Minimum is 1 byte, maximum is 65535 bytes
    - Padding format is: `[plaintext_length: u16][plaintext][zero_bytes]`
    - Padding algorithm is related to powers-of-two, with min padded msg size of 32 bytes
    - Plaintext length is encoded in big-endian as first 2 bytes of the padded blob
5. Encrypt padded content
    - Use ChaCha20, with key and nonce from step 3
6. Calculate MAC (message authentication code)
    - AAD (additional authenticated data) is used - instead of calculating MAC on ciphertext, it's calculated over a concatenation of `nonce` and `ciphertext`
    - Validate that AAD (nonce) is 32 bytes
7. Base64-encode (with padding) params using `concat(version, nonce, ciphertext, mac)`

Encrypted payloads MUST be included in an event's payload, hashed, and signed as defined in NIP 01, using schnorr signature scheme over secp256k1.

**Decryption**

Before decryption, the event's pubkey and signature MUST be validated as defined in NIP 01. The public key MUST be a valid non-zero secp256k1 curve point, and the signature must be valid secp256k1 schnorr signature. For exact validation rules, refer to BIP-340.

1. Check if first payload's character is `#`
    - `#` is an optional future-proof flag that means non-base64 encoding is used
    - The `#` is not present in base64 alphabet, but, instead of throwing `base64 is invalid`, implementations MUST indicate that the encryption version is not yet supported
2. Decode base64
    - Base64 is decoded into `version, nonce, ciphertext, mac`
    - If the version is unknown, implementations must indicate that the encryption version is not supported
    - Validate length of base64 message to prevent DoS on base64 decoder: it can be in range from 132 to 87472 chars
    - Validate length of decoded message to verify output of the decoder: it can be in range from 99 to 65603 bytes
3. Calculate conversation key
    - See step 1 of encryption
4. Calculate message keys
    - See step 3 of encryption
5. Calculate MAC (message authentication code) with AAD and compare
    - Stop and throw an error if MAC doesn't match the decoded one from step 2
    - Use constant-time comparison algorithm
6. Decrypt ciphertext
    - Use ChaCha20 with key and nonce from step 3
7. Remove padding
    - Read the first two BE bytes of plaintext that correspond to plaintext length
    - Verify that the length of sliced plaintext matches the value of the two BE bytes
    - Verify that calculated padding from step 3 of the encryption process matches the actual padding

**Details**

- Cryptographic methods

    - `secure_random_bytes(length)` fetches randomness from CSPRNG.
    - `hkdf(IKM, salt, info, L)` represents HKDF (RFC 5869) with SHA256 hash function comprised of methods `hkdf_extract(IKM, salt)` and `hkdf_expand(OKM, info, L)`.
    - `chacha20(key, nonce, data)` is ChaCha20 (RFC 8439) with starting counter set to 0.
    - `hmac_sha256(key, message)` is HMAC (RFC 2104) .
    - `secp256k1_ecdh(priv_a, pub_b)` is multiplication of point B by scalar a (`a · B`), defined in BIP340 . The operation produces a shared point, and we encode the shared point's 32-byte x coordinate, using method `bytes(P)` from BIP340. Private and public keys must be validated as per BIP340: pubkey must be a valid, on-curve point, and private key must be a scalar in range `[1, secp256k1_order - 1]`. NIP44 doesn't do hashing of the output: keep this in mind, because some libraries hash it using sha256. As an example, in libsecp256k1, unhashed version is available in `secp256k1_ec_pubkey_tweak_mul`
- Operators
    - `x[i:j]`, where `x` is a byte array and `i, j <= 0` returns a `(j - i)`-byte array with a copy of the `i`-th byte (inclusive) to the `j`-th byte (exclusive) of `x`.
- Constants `c`:
    - `min_plaintext_size` is 1. 1 byte msg is padded to 32 bytes.
    - `max_plaintext_size` is 65535 (64kB - 1). It is padded to 65536 bytes.
- Functions
    - `base64_encode(string)` and `base64_decode(bytes)` are Base64 (RFC 4648 , with padding)
    - `concat` refers to byte array concatenation
    - `is_equal_ct(a, b)` is constant-time equality check of 2 byte arrays
    - `utf8_encode(string)` and `utf8_decode(bytes)` transform string to byte array and back
    - `write_u8(number)` restricts number to values 0..255 and encodes into Big-Endian uint8 byte array
    - `write_u16_be(number)` restricts number to values 0..65535 and encodes into Big-Endian uint16 byte array
    - `zeros(length)` creates byte array of length `length >= 0`, filled with zeros
    - `floor(number)` and `log2(number)` are well-known mathematical methods

**Implementation pseudocode**

The following is a collection of python-like pseudocode functions which implement the above primitives, intended to guide implementers. A collection of implementations in different languages is available at https://github.com/paulmillr/nip44 .

```
## Calculates length of the padded byte array.
def calc_padded_len(unpadded_len):
  next_power = 1 << (floor(log2(unpadded_len - 1))) + 1
  if next_power <= 256:
    chunk = 32
  else:
    chunk = next_power / 8
  if unpadded_len <= 32:
    return 32
  else:
    return chunk * (floor((len - 1) / chunk) + 1)

## Converts unpadded plaintext to padded bytearray
def pad(plaintext):
  unpadded = utf8_encode(plaintext)
  unpadded_len = len(plaintext)
  if (unpadded_len < c.min_plaintext_size or
      unpadded_len > c.max_plaintext_size): raise Exception('invalid plaintext
length')
  prefix = write_u16_be(unpadded_len)
  suffix = zeros(calc_padded_len(unpadded_len) - unpadded_len)
  return concat(prefix, unpadded, suffix)

## Converts padded bytearray to unpadded plaintext
def unpad(padded):
  unpadded_len = read_uint16_be(padded[0:2])
  unpadded = padded[2:2+unpadded_len]
  if (unpadded_len == 0 or
      len(unpadded) != unpadded_len or
      len(padded) != 2 + calc_padded_len(unpadded_len)): raise Exception('invalid
padding')
```

```
  return utf8_decode(unpadded)

## metadata: always 65b (version: 1b, nonce: 32b, max: 32b)
## plaintext: 1b to 0xffff
## padded plaintext: 32b to 0xffff
## ciphertext: 32b+2 to 0xffff+2
## raw payload: 99 (65+32+2) to 65603 (65+0xffff+2)
## compressed payload (base64): 132b to 87472b
def decode_payload(payload):
  plen = len(payload)
  if plen == 0 or payload[0] == '#': raise Exception('unknown version')
  if plen < 132 or plen > 87472: raise Exception('invalid payload size')
  data = base64_decode(payload)
  dlen = len(d)
  if dlen < 99 or dlen > 65603: raise Exception('invalid data size');
  vers = data[0]
  if vers != 2: raise Exception('unknown version ' + vers)
  nonce = data[1:33]
  ciphertext = data[33:dlen - 32]
  mac = data[dlen - 32:dlen]
  return (nonce, ciphertext, mac)

def hmac_aad(key, message, aad):
  if len(aad) != 32: raise Exception('AAD associated data must be 32 bytes');
  return hmac(sha256, key, concat(aad, message));

## Calculates long-term key between users A and B: `get_key(Apriv, Bpub) ==
get_key(Bpriv, Apub)`
def get_conversation_key(private_key_a, public_key_b):
  shared_x = secp256k1_ecdh(private_key_a, public_key_b)
  return hkdf_extract(IKM=shared_x, salt=utf8_encode('nip44-v2'))

## Calculates unique per-message key
def get_message_keys(conversation_key, nonce):
  if len(conversation_key) != 32: raise Exception('invalid conversation_key length')
  if len(nonce) != 32: raise Exception('invalid nonce length')
  keys = hkdf_expand(OKM=conversation_key, info=nonce, L=76)
  chacha_key = keys[0:32]
  chacha_nonce = keys[32:44]
  hmac_key = keys[44:76]
  return (chacha_key, chacha_nonce, hmac_key)

def encrypt(plaintext, conversation_key, nonce):
  (chacha_key, chacha_nonce, hmac_key) = get_message_keys(conversation_key, nonce)
  padded = pad(plaintext)
  ciphertext = chacha20(key=chacha_key, nonce=chacha_nonce, data=padded)
  mac = hmac_aad(key=hmac_key, message=ciphertext, aad=nonce)
  return base64_encode(concat(write_u8(2), nonce, ciphertext, mac))

def decrypt(payload, conversation_key):
  (nonce, ciphertext, mac) = decode_payload(payload)
  (chacha_key, chacha_nonce, hmac_key) = get_message_keys(conversation_key, nonce)
  calculated_mac = hmac_aad(key=hmac_key, message=ciphertext, aad=nonce)
  if not is_equal_ct(calculated_mac, mac): raise Exception('invalid MAC')
  padded_plaintext = chacha20(key=chacha_key, nonce=chacha_nonce, data=ciphertext)
  return unpad(padded_plaintext)

## Usage:
##   conversation_key = get_conversation_key(sender_privkey, recipient_pubkey)
##   nonce = secure_random_bytes(32)
##   payload = encrypt('hello world', conversation_key, nonce)
##   'hello world' == decrypt(payload, conversation_key)
```

**Audit**

The v2 of the standard was audited by Cure53 in December 2023. Check out audit-2023.12.pdf and auditor's website
.

#### Tests and code

A collection of implementations in different languages is available at https://github.com/paulmillr/nip44 .

We publish extensive test vectors. Instead of having it in the document directly, a sha256 checksum of vectors is provided:

```
269ed0f69e4c192512cc779e78c555090cebc7c785b609e338a62afc3ce25040  nip44.vectors.json
```

Example of a test vector from the file:

```
{
  "sec1": "0000000000000000000000000000000000000000000000000000000000000001",
  "sec2": "0000000000000000000000000000000000000000000000000000000000000002",
  "conversation_key":
"c41c775356fd92eadc63ff5a0dc1da211b268cbea22316767095b2871ea1412d",
  "nonce": "0000000000000000000000000000000000000000000000000000000000000001",
  "plaintext": "a",
  "payload":
"AgAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABee0G5VSK0/9YypIObAtDKfYEAjD35uVkHyB0F4DwrcNaCXlCWZKaArsGr

}
```

The file also contains intermediate values. A quick guidance with regards to its usage:

- `valid.get_conversation_key`: calculate conversation_key from secret key sec1 and public key pub2
- `valid.get_message_keys`: calculate chacha_key, chacha_nonce, hmac_key from conversation_key and nonce
- `valid.calc_padded_len`: take unpadded length (first value), calculate padded length (second value)
- `valid.encrypt_decrypt`: emulate real conversation. Calculate pub2 from sec2, verify conversation_key from (sec1, pub2), encrypt, verify payload, then calculate pub1 from sec1, verify conversation_key from (sec2, pub1), decrypt, verify plaintext.
- `valid.encrypt_decrypt_long_msg`: same as previous step, but instead of a full plaintext and payload, their checksum is provided.
- `invalid.encrypt_msg_lengths`
- `invalid.get_conversation_key`: calculating conversation_key must throw an error
- `invalid.decrypt`: decrypting message content must throw an error

## NIP-45

## Event Counts

`draft` `optional`

Relays may support the verb `COUNT`, which provides a mechanism for obtaining event counts.

### Motivation

Some queries a client may want to execute against connected relays are prohibitively expensive, for example, in order to retrieve follower counts for a given pubkey, a client must query all kind-3 events referring to a given pubkey only to count them. The result may be cached, either by a client or by a separate indexing server as an alternative, but both options erode the decentralization of the network by creating a second-layer protocol on top of Nostr.

### Filters and return values

This NIP defines the verb `COUNT`, which accepts a subscription id and filters as specified in NIP 01 for the verb `REQ`. Multiple filters are OR'd together and aggregated into a single count result.

```
["COUNT", <subscription_id>, <filters JSON>...]
```

Counts are returned using a `COUNT` response in the form `{"count": <integer>}`. Relays may use probabilistic counts to reduce compute requirements. In case a relay uses probabilistic counts, it MAY indicate it in the response with `approximate` key i.e. `{"count": <integer>, "approximate": <true|false>}`.

```
["COUNT", <subscription_id>, {"count": <integer>}]
```

Whenever the relay decides to refuse to fulfill the `COUNT` request, it MUST return a `CLOSED` message.

### Examples

#### Followers count

```
["COUNT", <subscription_id>, {"kinds": [3], "#p": [<pubkey>]}]
["COUNT", <subscription_id>, {"count": 238}]
```

#### Count posts and reactions

```
["COUNT", <subscription_id>, {"kinds": [1, 7], "authors": [<pubkey>]}]
["COUNT", <subscription_id>, {"count": 5}]
```

#### Count posts approximately

```
["COUNT", <subscription_id>, {"kinds": [1]}]
["COUNT", <subscription_id>, {"count": 93412452, "approximate": true}]
```

#### Relay refuses to count

```
["COUNT", <subscription_id>, {"kinds": [4], "authors": [<pubkey>], "#p":
[<pubkey>]}]
["CLOSED", <subscription_id>, "auth-required: cannot count other people's DMs"]
```

## NIP-46

## Nostr Remote Signing

### Changes

`remote-signer-key` is introduced, passed in bunker url, clients must differentiate between `remote-signer-pubkey` and `user-pubkey`, must call `get_public_key` after connect, nip05 login is removed, create_account moved to another NIP.

### Rationale

Private keys should be exposed to as few systems - apps, operating systems, devices - as possible as each system adds to the attack surface.

This NIP describes a method for 2-way communication between a remote signer and a Nostr client. The remote signer could be, for example, a hardware device dedicated to signing Nostr events, while the client is a normal Nostr client.

### Terminology

- **user**: A person that is trying to use Nostr.
- **client**: A user-facing application that *user* is looking at and clicking buttons in. This application will send requests to *remote-signer*.
- **remote-signer**: A daemon or server running somewhere that will answer requests from *client*, also known as “bunker”.
- **client-keypair/pubkey**: The keys generated by *client*. Used to encrypt content and communicate with *remote-signer*.
- **remote-signer-keypair/pubkey**: The keys used by *remote-signer* to encrypt content and communicate with *client*. This keypair MAY be same as *user-keypair*, but not necessarily.
- **user-keypair/pubkey**: The actual keys representing *user* (that will be used to sign events in response to `sign_event` requests, for example). The *remote-signer* generally has control over these keys.

All pubkeys specified in this NIP are in hex format.

### Overview

1. *client* generates `client-keypair`. This keypair doesn’t need to be communicated to *user* since it’s largely disposable. *client* might choose to store it locally and they should delete it on logout;
2. A connection is established (see below), *remote-signer* learns `client-pubkey`, *client* learns `remote-signer-pubkey`.
3. *client* uses `client-keypair` to send requests to *remote-signer* by `p`-tagging and encrypting to `remote-signer-pubkey`;
4. *remote-signer* responds to *client* by `p`-tagging and encrypting to the `client-pubkey`.
5. *client* requests `get_public_key` to learn `user-pubkey`.

### Initiating a connection

There are two ways to initiate a connection:

**Direct connection initiated by *remote-signer***

*remote-signer* provides connection token in the form:

```
bunker://<remote-signer-pubkey>?relay=<wss://relay-to-connect-on>&relay=
<wss://another-relay-to-connect-on>&secret=<optional-secret-value>
```

*user* passes this token to *client*, which then sends `connect` request to *remote-signer* via the specified relays. Optional secret can be used for single successfully established connection only, *remote-signer* SHOULD ignore new attempts to establish connection with old secret.

**Direct connection initiated by the *client***

*client* provides a connection token using `nostrconnect://` as the protocol, and `client-pubkey` as the origin. Additional information should be passed as query parameters:

- `relay` (required) - one or more relay urls on which the *client* is listening for responses from the *remote-signer*.
- `secret` (required) - a short random string that the *remote-signer* should return as the `result` field of its response.
- `perms` (optional) - a comma-separated list of permissions the *client* is requesting be approved by the *remote-signer*
- `name` (optional) - the name of the *client* application
- `url` (optional) - the canonical url of the *client* application
- `image` (optional) - a small image representing the *client* application

Here's an example:

```
nostrconnect://83f3b2ae6aa368e8275397b9c26cf550101d63ebaab900d19dd4a4429f5ad8f5?
relay=wss%3A%2F%2Frelay1.example.com&perms=nip44_encrypt%2Cnip44_decrypt%2Csign_event%3A13%2Csign_event
```

*user* passes this token to *remote-signer*, which then sends `connect` *response* event to the `client-pubkey` via the specified relays. Client discovers `remote-signer-pubkey` from connect response author. `secret` value MUST be provided to avoid connection spoofing, *client* MUST validate the `secret` returned by `connect` response.

### Request Events `kind: 24133`

```
{
    "kind": 24133,
    "pubkey": <local_keypair_pubkey>,
    "content": <nip44(<request>)>,
    "tags": [["p", <remote-signer-pubkey>]],
}
```

The `content` field is a JSON-RPC-like message that is NIP-44 encrypted and has the following structure:

```
{
    "id": <random_string>,
    "method": <method_name>,
    "params": [array_of_strings]
}
```

- `id` is a random string that is a request ID. This same ID will be sent back in the response payload.
- `method` is the name of the method/command (detailed below).
- `params` is a positional array of string parameters.

**Methods/Commands**

Each of the following are methods that the *client* sends to the *remote-signer*.

| Command | Params | Result |
|---|---|---|
| `connect` | `[<remote-signer-pubkey>, <optional_secret>, <optional_requested_permissions>]` | "ack" OR `<required-secret-value>` |
| `sign_event` | `[<{kind, content, tags, created_at}>]` | `json_stringified(<signed_event>)` |
| `ping` | `[]` | "pong" |
| `get_relays` | `[]` | `json_stringified({<relay_url>: {read: <boolean>, write: <boolean>}})` |
| `get_public_key` | `[]` | `<user-pubkey>` |

| Command | Params | Result |
| --- | --- | --- |
| `nip04_encrypt` | `[<third_party_pubkey>, <plaintext_to_encrypt>]` | `<nip04_ciphertext>` |
| `nip04_decrypt` | `[<third_party_pubkey>, <nip04_ciphertext_to_decrypt>]` | `<plaintext>` |
| `nip44_encrypt` | `[<third_party_pubkey>, <plaintext_to_encrypt>]` | `<nip44_ciphertext>` |
| `nip44_decrypt` | `[<third_party_pubkey>, <nip44_ciphertext_to_decrypt>]` | `<plaintext>` |

#### Requested permissions

The `connect` method may be provided with `optional_requested_permissions` for user convenience. The permissions are a comma-separated list of `method[:params]`, i.e. `nip44_encrypt,sign_event:4` meaning permissions to call `nip44_encrypt` and to call `sign_event` with `kind:4`. Optional parameter for `sign_event` is the kind number, parameters for other methods are to be defined later. Same permission format may be used for `perms` field of `metadata` in `nostrconnect://` string.

### Response Events `kind:24133`

```
{
    "id": <id>,
    "kind": 24133,
    "pubkey": <remote-signer-pubkey>,
    "content": <nip44(<response>)>,
    "tags": [["p", <client-pubkey>]],
    "created_at": <unix timestamp in seconds>
}
```

The `content` field is a JSON-RPC-like message that is NIP-44 encrypted and has the following structure:

```
{
    "id": <request_id>,
    "result": <results_string>,
    "error": <optional_error_string>
}
```

- `id` is the request ID that this response is for.
- `results` is a string of the result of the call (this can be either a string or a JSON stringified object)
- `error`, *optionally*, it is an error in string form, if any. Its presence indicates an error with the request.

### Example flow for signing an event

- `remote-signer-pubkey` is `fa984bd7dbb282f07e16e7ae87b26a2a7b9b90b7246a44771f0cf5ae58018f52`
- `user-pubkey` is also `fa984bd7dbb282f07e16e7ae87b26a2a7b9b90b7246a44771f0cf5ae58018f52`
- `client-pubkey` is `eff37350d839ce3707332348af4549a96051bd695d3223af4aabce4993531d86`

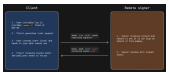
#### Signature request

```
{
    "kind": 24133,
    "pubkey": "eff37350d839ce3707332348af4549a96051bd695d3223af4aabce4993531d86",
    "content": nip44({
        "id": <random_string>,
        "method": "sign_event",
        "params": [json_stringified(<{
            content: "Hello, I'm signing remotely",
            kind: 1,
            tags: [],
            created_at: 1714078911
        }>)]
    }),
    "tags": [["p",
"fa984bd7dbb282f07e16e7ae87b26a2a7b9b90b7246a44771f0cf5ae58018f52"]], // p-tags the
remote-signer-pubkey
}
```

#### Response event

```
{
    "kind": 24133,
    "pubkey": "fa984bd7dbb282f07e16e7ae87b26a2a7b9b90b7246a44771f0cf5ae58018f52",
    "content": nip44({
        "id": <random_string>,
        "result": json_stringified(<signed-event>)
    }),
    "tags": [["p",
"eff37350d839ce3707332348af4549a96051bd695d3223af4aabce4993531d86"]], // p-tags the
client-pubkey
}
```

**Diagram**



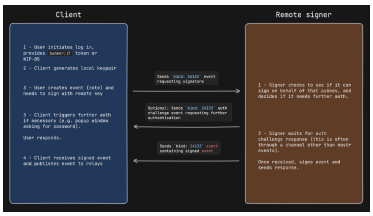
## Auth Challenges

An Auth Challenge is a response that a *remote-signer* can send back when it needs the *user* to authenticate via other means. The response `content` object will take the following form:

```
{
    "id": <request_id>,
    "result": "auth_url",
    "error": <URL_to_display_to_end_user>
}
```

*client* should display (in a popup or new tab) the URL from the `error` field and then subscribe/listen for another response from the *remote-signer* (reusing the same request ID). This event will be sent once the user authenticates in the other window (or will never arrive if the user doesn't authenticate).

**Example event signing request with auth challenge**




## Appendix

**Announcing *remote-signer* metadata**

*remote-signer* MAY publish it's metadata by using NIP-05 and NIP-89 . With NIP-05, a request to `<remote-signer>/.well-known/nostr.json?name=_` MAY return this:

```
{
    "names":{
        "_": <remote-signer-app-pubkey>,
    },
    "nip46": {
        "relays": ["wss://relay1","wss://relay2"...],
        "nostrconnect_url": "https://remote-signer-domain.example/<nostrconnect>"
    }
}
```

The `<remote-signer-app-pubkey>` MAY be used to verify the domain from *remote-signer*'s NIP-89 event (see below). `relays` SHOULD be used to construct a more precise `nostrconnect://` string for the specific `remote-signer`. `nostrconnect_url` template MAY be used to redirect users to *remote-signer*'s connection flow by replacing `<nostrconnect>` placeholder with an actual `nostrconnect://` string.

**Remote signer discovery via NIP-89**

*remote-signer* MAY publish a NIP-89 `kind: 31990` event with `k` tag of `24133`, which MAY also include one or more `relay` tags and MAY include `nostrconnect_url` tag. The semantics of `relay` and `nostrconnect_url` tags are

the same as in the section above.

*client* MAY improve UX by discovering *remote-signers* using their `kind: 31990` events. *client* MAY then pre-generate `nostrconnect://` strings for the *remote-signers*, and SHOULD in that case verify that `kind: 31990` event's author is mentioned in signer's `nostr.json?name=_` file as `<remote-signer-app-pubkey>`.

## NIP-47

## Nostr Wallet Connect

`draft` `optional`

### Rationale

This NIP describes a way for clients to access a remote lightning wallet through a standardized protocol. Custodians may implement this, or the user may run a bridge that bridges their wallet/node and the Nostr Wallet Connect protocol.

### Terms

- **client**: Nostr app on any platform that wants to interact with a lightning wallet.
- **user**: The person using the **client**, and wants to connect their wallet to their **client**.
- **wallet service**: Nostr app that typically runs on an always-on computer (eg. in the cloud or on a Raspberry Pi). This app has access to the APIs of the wallets it serves.

### Theory of Operation

1. **Users** who wish to use this NIP to allow **client(s)** to interact with their wallet must first acquire a special "connection" URI from their NIP-47 compliant wallet application. The wallet application may provide this URI using a QR screen, or a pasteable string, or some other means.

2. The **user** should then copy this URI into their **client(s)** by pasting, or scanning the QR, etc. The **client(s)** should save this URI and use it later whenever the **user** (or the **client** on the user's behalf) wants to interact with the wallet. The **client** should then request an `info` (13194) event from the relay(s) specified in the URI. The **wallet service** will have sent that event to those relays earlier, and the relays will hold it as a replaceable event.

3. When the **user** initiates a payment their nostr **client** create a `pay_invoice` request, encrypts it using a token from the URI, and sends it (kind 23194) to the relay(s) specified in the connection URI. The **wallet service** will be listening on those relays and will decrypt the request and then contact the **user's** wallet application to send the payment. The **wallet service** will know how to talk to the wallet application because the connection URI specified relay(s) that have access to the wallet app API.

4. Once the payment is complete the **wallet service** will send an encrypted `response` (kind 23195) to the **user** over the relay(s) in the URI.

5. The **wallet service** may send encrypted notifications (kind 23196) of wallet events (such as a received payment) to the **client**.

### Events

There are four event kinds:

- `NIP-47 info event`: 13194
- `NIP-47 request`: 23194
- `NIP-47 response`: 23195
- `NIP-47 notification event`: 23196

#### Info Event

The info event should be a replaceable event that is published by the **wallet service** on the relay to indicate which capabilities it supports.

The content should be a plaintext string with the supported capabilities space-separated, eg. `pay_invoice get_balance notifications`.

If the **wallet service** supports notifications, the info event SHOULD contain a `notifications` tag with the supported notification types space-separated, eg. `payment_received payment_sent`.

#### Request and Response Events

Both the request and response events SHOULD contain one `p` tag, containing the public key of the **wallet service** if this is a request, and the public key of the **user** if this is a response. The response event SHOULD contain an `e` tag with the id of the request event it is responding to. Optionally, a request can have an `expiration` tag that has a unix timestamp in seconds. If the request is received after this timestamp, it should be ignored.

The content of requests and responses is encrypted with NIP04 , and is a JSON-RPCish object with a semi-fixed structure:

Request:

```
{
    "method": "pay_invoice", // method, string
    "params": { // params, object
        "invoice": "lnbc50n1..." // command-related data
    }
}
```

Response:

```
{
    "result_type": "pay_invoice", //indicates the structure of the result field
    "error": { //object, non-null in case of error
        "code": "UNAUTHORIZED", //string error code, see below
        "message": "human readable error message"
    },
    "result": { // result, object. null in case of error.
        "preimage": "0123456789abcdef..." // command-related data
    }
}
```

The `result_type` field MUST contain the name of the method that this event is responding to. The `error` field MUST contain a `message` field with a human readable error message and a `code` field with the error code if the command was not successful. If the command was successful, the `error` field must be null.

#### Notification Events

The notification event SHOULD contain one `p` tag, the public key of the **user**.

The content of notifications is encrypted with NIP04 , and is a JSON-RPCish object with a semi-fixed structure:

```
{
    "notification_type": "payment_received", //indicates the structure of the
notification field
    "notification": {
        "payment_hash": "0123456789abcdef..." // notification-related data
    }
}
```

#### Error codes

- `RATE_LIMITED`: The client is sending commands too fast. It should retry in a few seconds.
- `NOT_IMPLEMENTED`: The command is not known or is intentionally not implemented.
- `INSUFFICIENT_BALANCE`: The wallet does not have enough funds to cover a fee reserve or the payment amount.
- `QUOTA_EXCEEDED`: The wallet has exceeded its spending quota.
- `RESTRICTED`: This public key is not allowed to do this operation.
- `UNAUTHORIZED`: This public key has no wallet connected.
- `INTERNAL`: An internal error.
- `OTHER`: Other error.

### Nostr Wallet Connect URI

**client** discovers **wallet service** by scanning a QR code, handling a deeplink or pasting in a URI.

The **wallet service** generates this connection URI with protocol `nostr+walletconnect://` and base path it's hex-encoded `pubkey` with the following query string parameters:

- `relay` Required. URL of the relay where the **wallet service** is connected and will be listening for events. May be more than one.
- `secret` Required. 32-byte randomly generated hex encoded string. The **client** MUST use this to sign events and encrypt payloads when communicating with the **wallet service**.
    - Authorization does not require passing keys back and forth.
    - The user can have different keys for different applications. Keys can be revoked and created at will and have arbitrary constraints (eg. budgets).
    - The key is harder to leak since it is not shown to the user and backed up.
    - It improves privacy because the user's main key would not be linked to their payments.
- `lud16` Recommended. A lightning address that clients can use to automatically setup the `lud16` field on the user's profile if they have none configured.

The **client** should then store this connection and use it when the user wants to perform actions like paying an invoice. Due to this NIP using ephemeral events, it is recommended to pick relays that do not close connections on inactivity to not drop events.

**Example connection string**

```
nostr+walletconnect://b889ff5b1513b641e2a139f661a661364979c5beee91842f8f0ef42ab558e9d4?
relay=wss%3A%2F%2Frelay.damus.io&secret=71a8c14c1407c113601079c4302dab36460f0ccd0ad506f1f2dc73b5100e4f
```

## Commands

**`pay_invoice`**

Description: Requests payment of an invoice.

Request:

```
{
    "method": "pay_invoice",
    "params": {
        "invoice": "lnbc50n1...", // bolt11 invoice
        "amount": 123, // invoice amount in msats, optional
    }
}
```

Response:

```
{
    "result_type": "pay_invoice",
    "result": {
        "preimage": "0123456789abcdef...", // preimage of the payment
        "fees_paid": 123, // value in msats, optional
    }
}
```

Errors:

- `PAYMENT_FAILED`: The payment failed. This may be due to a timeout, exhausting all routes, insufficient capacity or similar.

**`multi_pay_invoice`**

Description: Requests payment of multiple invoices.

Request:

```
{
    "method": "multi_pay_invoice",
    "params": {
        "invoices": [
          {"id":"4da52c32a1", "invoice": "lnbc1...", "amount": 123}, // bolt11
invoice and amount in msats, amount is optional
          {"id":"3da52c32a1", "invoice": "lnbc50n1..."},
        ],
    }
}
```

Response:

For every invoice in the request, a separate response event is sent. To differentiate between the responses, each response event contains a `d` tag with the id of the invoice it is responding to, if no id was given, then the payment hash of the invoice should be used.

```
{
    "result_type": "multi_pay_invoice",
    "result": {
        "preimage": "0123456789abcdef...", // preimage of the payment
        "fees_paid": 123, // value in msats, optional
    }
}
```

Errors:

- `PAYMENT_FAILED`: The payment failed. This may be due to a timeout, exhausting all routes, insufficient capacity or similar.

**`pay_keysend`**

Request:

```
{
    "method": "pay_keysend",
    "params": {
        "amount": 123, // invoice amount in msats, required
        "pubkey": "03...", // payee pubkey, required
        "preimage": "0123456789abcdef...", // preimage of the payment, optional
        "tlv_records": [ // tlv records, optional
            {
                "type": 5482373484, // tlv type
                "value": "0123456789abcdef" // hex encoded tlv value
            }
        ]
    }
}
```

Response:

```
{
    "result_type": "pay_keysend",
    "result": {
        "preimage": "0123456789abcdef...", // preimage of the payment
        "fees_paid": 123, // value in msats, optional
    }
}
```

Errors:

- `PAYMENT_FAILED`: The payment failed. This may be due to a timeout, exhausting all routes, insufficient capacity or similar.

**`multi_pay_keysend`**

Description: Requests multiple keysend payments.

Has an array of keysends, these follow the same semantics as `pay_keysend`, just done in a batch

Request:

```
{
    "method": "multi_pay_keysend",
    "params": {
        "keysends": [
          {"id": "4c5b24a351", "pubkey": "03...", "amount": 123},
          {"id": "3da52c32a1", "pubkey": "02...", "amount": 567, "preimage":
"abc123..", "tlv_records": [{"type": 696969, "value":
"77616c5f6872444873305242454d353736"}]},
```

```
        ],
    }
}
```

Response:

For every keysend in the request, a separate response event is sent. To differentiate between the responses, each response event contains an `d` tag with the id of the keysend it is responding to, if no id was given, then the pubkey should be used.

```
{
    "result_type": "multi_pay_keysend",
    "result": {
        "preimage": "0123456789abcdef...", // preimage of the payment
        "fees_paid": 123, // value in msats, optional
    }
}
```

Errors:

- `PAYMENT_FAILED`: The payment failed. This may be due to a timeout, exhausting all routes, insufficient capacity or similar.

**`make_invoice`**

Request:

```
{
    "method": "make_invoice",
    "params": {
        "amount": 123, // value in msats
        "description": "string", // invoice's description, optional
        "description_hash": "string", // invoice's description hash, optional
        "expiry": 213 // expiry in seconds from time invoice is created, optional
    }
}
```

Response:

```
{
    "result_type": "make_invoice",
    "result": {
        "type": "incoming", // "incoming" for invoices, "outgoing" for payments
        "invoice": "string", // encoded invoice, optional
        "description": "string", // invoice's description, optional
        "description_hash": "string", // invoice's description hash, optional
        "preimage": "string", // payment's preimage, optional if unpaid
        "payment_hash": "string", // Payment hash for the payment
        "amount": 123, // value in msats
        "fees_paid": 123, // value in msats
        "created_at": unixtimestamp, // invoice/payment creation time
        "expires_at": unixtimestamp, // invoice expiration time, optional if not
applicable
        "metadata": {} // generic metadata that can be used to add things like
zap/boostagram details for a payer name/comment/etc.
    }
}
```

**`lookup_invoice`**

Request:

```
{
    "method": "lookup_invoice",
    "params": {
        "payment_hash": "31afdf1..", // payment hash of the invoice, one of
payment_hash or invoice is required
        "invoice": "lnbc50n1..." // invoice to lookup
```

```
    }
}
```

Response:

```
{
    "result_type": "lookup_invoice",
    "result": {
        "type": "incoming", // "incoming" for invoices, "outgoing" for payments
        "invoice": "string", // encoded invoice, optional
        "description": "string", // invoice's description, optional
        "description_hash": "string", // invoice's description hash, optional
        "preimage": "string", // payment's preimage, optional if unpaid
        "payment_hash": "string", // Payment hash for the payment
        "amount": 123, // value in msats
        "fees_paid": 123, // value in msats
        "created_at": unixtimestamp, // invoice/payment creation time
        "expires_at": unixtimestamp, // invoice expiration time, optional if not
applicable
        "settled_at": unixtimestamp, // invoice/payment settlement time, optional if
unpaid
        "metadata": {} // generic metadata that can be used to add things like
zap/boostagram details for a payer name/comment/etc.
    }
}
```

Errors:

- `NOT_FOUND`: The invoice could not be found by the given parameters.

**`list_transactions`**

Lists invoices and payments. If `type` is not specified, both invoices and payments are returned. The `from` and `until` parameters are timestamps in seconds since epoch. If `from` is not specified, it defaults to 0. If `until` is not specified, it defaults to the current time. Transactions are returned in descending order of creation time.

Request:

```
{
    "method": "list_transactions",
    "params": {
        "from": 1693876973, // starting timestamp in seconds since epoch
(inclusive), optional
        "until": 1703225078, // ending timestamp in seconds since epoch (inclusive),
optional
        "limit": 10, // maximum number of invoices to return, optional
        "offset": 0, // offset of the first invoice to return, optional
        "unpaid": true, // include unpaid invoices, optional, default false
        "type": "incoming", // "incoming" for invoices, "outgoing" for payments,
undefined for both
    }
}
```

Response:

```
{
    "result_type": "list_transactions",
    "result": {
        "transactions": [
            {
               "type": "incoming", // "incoming" for invoices, "outgoing" for
payments
               "invoice": "string", // encoded invoice, optional
               "description": "string", // invoice's description, optional
               "description_hash": "string", // invoice's description hash, optional
               "preimage": "string", // payment's preimage, optional if unpaid
               "payment_hash": "string", // Payment hash for the payment
               "amount": 123, // value in msats
```

```
                "fees_paid": 123, // value in msats
                "created_at": unixtimestamp, // invoice/payment creation time
                "expires_at": unixtimestamp, // invoice expiration time, optional if
not applicable
                "settled_at": unixtimestamp, // invoice/payment settlement time,
optional if unpaid
                "metadata": {} // generic metadata that can be used to add things
like zap/boostagram details for a payer name/comment/etc.
            }
        ],
    },
}
```

**`get_balance`**

Request:

```
{
    "method": "get_balance",
    "params": {}
}
```

Response:

```
{
    "result_type": "get_balance",
    "result": {
        "balance": 10000, // user's balance in msats
    }
}
```

**`get_info`**

Request:

```
{
    "method": "get_info",
    "params": {}
}
```

Response:

```
{
    "result_type": "get_info",
    "result": {
            "alias": "string",
            "color": "hex string",
            "pubkey": "hex string",
            "network": "string", // mainnet, testnet, signet, or regtest
            "block_height": 1,
            "block_hash": "hex string",
            "methods": ["pay_invoice", "get_balance", "make_invoice",
"lookup_invoice", "list_transactions", "get_info"], // list of supported methods for
this connection
            "notifications": ["payment_received", "payment_sent"], // list of
supported notifications for this connection, optional.
    }
}
```

### Notifications

**`payment_received`**

Description: A payment was successfully received by the wallet.

Notification:

```
{
    "notification_type": "payment_received",
    "notification": {
        "type": "incoming",
        "invoice": "string", // encoded invoice
        "description": "string", // invoice's description, optional
        "description_hash": "string", // invoice's description hash, optional
        "preimage": "string", // payment's preimage
        "payment_hash": "string", // Payment hash for the payment
        "amount": 123, // value in msats
        "fees_paid": 123, // value in msats
        "created_at": unixtimestamp, // invoice/payment creation time
        "expires_at": unixtimestamp, // invoice expiration time, optional if not
applicable
        "settled_at": unixtimestamp, // invoice/payment settlement time
        "metadata": {} // generic metadata that can be used to add things like
zap/boostagram details for a payer name/comment/etc.
    }
}
```

**`payment_sent`**

Description: A payment was successfully sent by the wallet.

Notification:

```
{
    "notification_type": "payment_sent",
    "notification": {
        "type": "outgoing",
        "invoice": "string", // encoded invoice
        "description": "string", // invoice's description, optional
        "description_hash": "string", // invoice's description hash, optional
        "preimage": "string", // payment's preimage
        "payment_hash": "string", // Payment hash for the payment
        "amount": 123, // value in msats
        "fees_paid": 123, // value in msats
        "created_at": unixtimestamp, // invoice/payment creation time
        "expires_at": unixtimestamp, // invoice expiration time, optional if not
applicable
        "settled_at": unixtimestamp, // invoice/payment settlement time
        "metadata": {} // generic metadata that can be used to add things like
zap/boostagram details for a payer name/comment/etc.
    }
}
```

### Example pay invoice flow

0. The user scans the QR code generated by the **wallet service** with their **client** application, they follow a `nostr+walletconnect://` deeplink or configure the connection details manually.
1. **client** sends an event to the **wallet service** with kind `23194`. The content is a `pay_invoice` request. The private key is the secret from the connection string above.
2. **wallet service** verifies that the author's key is authorized to perform the payment, decrypts the payload and sends the payment.
3. **wallet service** responds to the event by sending an event with kind `23195` and content being a response either containing an error message or a preimage.

### Using a dedicated relay

This NIP does not specify any requirements on the type of relays used. However, if the user is using a custodial service it might make sense to use a relay that is hosted by the custodial service. The relay may then enforce authentication to prevent metadata leaks. Not depending on a 3rd party relay would also improve reliability in this case.

### Appendix

#### Example NIP-47 info event

```
{
  "id": "df467db0a9f9ec77ffe6f561811714ccaa2e26051c20f58f33c3d66d6c2b4d1c",
  "pubkey": "c04ccd5c82fc1ea3499b9c6a5c0a7ab627fbe00a0116110d4c750faeaecba1e2",
  "created_at": 1713883677,
  "kind": 13194,
  "tags": [
    [
      "notifications",
      "payment_received payment_sent"
    ]
  ],
  "content": "pay_invoice pay_keysend get_balance get_info make_invoice
lookup_invoice list_transactions multi_pay_invoice multi_pay_keysend sign_message
notifications",
  "sig":
"31f57b369459b5306a5353aa9e03be7fbde169bc881c3233625605dd12f53548179def16b9fe1137e6465d7e4d5bb27ce81fd

}
```


## NIP-48

## Proxy Tags

`draft` `optional`

Nostr events bridged from other protocols such as ActivityPub can link back to the source object by including a `"proxy"` tag, in the form:

```
["proxy", <id>, <protocol>]
```

Where:

- `<id>` is the ID of the source object. The ID format varies depending on the protocol. The ID must be universally unique, regardless of the protocol.
- `<protocol>` is the name of the protocol, e.g. `"activitypub"`.

Clients may use this information to reconcile duplicated content bridged from other protocols, or to display a link to the source object.

Proxy tags may be added to any event kind, and doing so indicates that the event did not originate on the Nostr protocol, and instead originated elsewhere on the web.

#### Supported protocols

This list may be extended in the future.

| Protocol | ID format | Example |
|---|---|---|
| `activitypub` | URL | `https://gleasonator.com/objects/9f524868-c1a0-4ee7-ad51-aaa23d68b526` |
| `atproto` | AT URI | `at://did:plc:zhbjlbmir5dganqhueg7y4i3/app.bsky.feed.post/3jt5hlibeol2i` |
| `rss` | URL with guid fragment | `https://soapbox.pub/rss/feed.xml#https%3A%2F%2Fsoapbox.pub%2Fblog%2Fmostr-fediverse-nostr-bridge` |
| `web` | URL | `https://twitter.com/jack/status/20` |

#### Examples

ActivityPub object:


```
{
  "kind": 1,
  "content": "I'm vegan btw",
  "tags": [
    [
      "proxy",
      "https://gleasonator.com/objects/8f6fac53-4f66-4c6e-ac7d-92e5e78c3e79",
      "activitypub"
    ]
  ],
  "pubkey": "79c2cae114ea28a981e7559b4fe7854a473521a8d22a66bbab9fa248eb820ff6",
  "created_at": 1691091365,
```

```
  "id": "55920b758b9c7b17854b6e3d44e6a02a83d1cb49e1227e75a30426dea94d4cb2",
  "sig":
"a72f12c08f18e85d98fb92ae89e2fe63e48b8864c5e10fbdd5335f3c9f936397a6b0a7350efe251f8168b1601d7012d4a6d0e

}
```

**See also**

- FEP-fffd: Proxy Objects
- Mostr bridge

## NIP-49

## Private Key Encryption

`draft` `optional`

This NIP defines a method by which clients can encrypt (and decrypt) a user's private key with a password.

## Symmetric Encryption Key derivation

PASSWORD = Read from the user. The password should be unicode normalized to NFKC format to ensure that the password can be entered identically on other computers/clients.

LOG_N = Let the user or implementer choose one byte representing a power of 2 (e.g. 18 represents 262,144) which is used as the number of rounds for scrypt. Larger numbers take more time and more memory, and offer better protection:

```
| LOG_N | MEMORY REQUIRED | APPROX TIME ON FAST COMPUTER |
|-------|-----------------|------------------------------ |
| 16    | 64 MiB          | 100 ms                       |
| 18    | 256 MiB         |                              |
| 20    | 1 GiB           | 2 seconds                    |
| 21    | 2 GiB           |                              |
| 22    | 4 GiB           |                              |
```

SALT = 16 random bytes

SYMMETRIC_KEY = scrypt(password=PASSWORD, salt=SALT, log_n=LOG_N, r=8, p=1)

The symmetric key should be 32 bytes long.

This symmetric encryption key is temporary and should be zeroed and discarded after use and not stored or reused for any other purpose.

## Encrypting a private key

The private key encryption process is as follows:

PRIVATE_KEY = User's private (secret) secp256k1 key as 32 raw bytes (not hex or bech32 encoded!)

KEY_SECURITY_BYTE = one of:

- 0x00 - if the key has been known to have been handled insecurely (stored unencrypted, cut and paste unencrypted, etc)
- 0x01 - if the key has NOT been known to have been handled insecurely (stored unencrypted, cut and paste unencrypted, etc)
- 0x02 - if the client does not track this data

ASSOCIATED_DATA = KEY_SECURITY_BYTE

NONCE = 24 byte random nonce

CIPHERTEXT = XChaCha20-Poly1305( plaintext=PRIVATE_KEY, associated_data=ASSOCIATED_DATA, nonce=NONCE, key=SYMMETRIC_KEY )

VERSION_NUMBER = 0x02

CIPHERTEXT_CONCATENATION = concat( VERSION_NUMBER, LOG_N, SALT, NONCE, ASSOCIATED_DATA, CIPHERTEXT )

ENCRYPTED_PRIVATE_KEY = bech32_encode('ncryptsec', CIPHERTEXT_CONCATENATION)

The output prior to bech32 encoding should be 91 bytes long.

The decryption process operates in the reverse.

## Test Data

### Password Unicode Normalization

The following password input: “ÅΩḯ”

- Unicode Codepoints: U+212B U+2126 U+1E9B U+0323
- UTF-8 bytes: [0xE2, 0x84, 0xAB, 0xE2, 0x84, 0xA6, 0xE1, 0xBA, 0x9B, 0xCC, 0xA3]

Should be converted into the unicode normalized NFKC format prior to use in scrypt: “ÅΩḯ”

- Unicode Codepoints: U+00C5 U+03A9 U+1E69
- UTF-8 bytes: [0xC3, 0x85, 0xCE, 0xA9, 0xE1, 0xB9, 0xA9]

### Encryption

The encryption process is non-deterministic due to the random nonce.

### Decryption

The following encrypted private key:

`ncryptsec1qgg9947rlpvqu76pj5ecreduf9jxhselq2nae2kghhvd5g7dgjtcxfqtd67p9m0w57lspw8gsq6yphnm8623nsl8xn9j4`

When decrypted with password=‘nostr’ and log_n=16 yields the following hex-encoded private key:

`3501454135014541350145413501453fefb02227e449e57cf4d3a3ce05378683`

## Discussion

#### On Key Derivation

Passwords make poor cryptographic keys. Prior to use as a cryptographic key, two things need to happen:

1. An encryption key needs to be deterministically created from the password such that is has a uniform functionally random distribution of bits, such that the symmetric encryption algorithm’s assumptions are valid, and
2. A slow irreversible algorithm should be injected into the process, so that brute-force attempts to decrypt by trying many passwords are severely hampered.

These are achieved using a password-based key derivation function. We use scrypt, which has been proven to be maximally memory hard and which several cryptographers have indicated to the author is better than argon2 even though argon2 won a competition in 2015.

#### On the symmetric encryption algorithm

XChaCha20-Poly1305 is typically favored by cryptographers over AES and is less associated with the U.S. government. It (or it’s earlier variant without the ‘X’) is gaining wide usage, is used in TLS and OpenSSH, and is available in most modern crypto libraries.

## Recommendations

It is not recommended that users publish these encrypted private keys to nostr, as cracking a key may become easier when an attacker can amass many encrypted private keys.

It is recommended that clients zero out the memory of passwords and private keys before freeing that memory.

## NIP-50

## Search Capability

`draft` `optional`

### Abstract

Many Nostr use cases require some form of general search feature, in addition to structured queries by tags or ids. Specifics of the search algorithms will differ between event kinds, this NIP only describes a general extensible framework for performing such queries.

### `search` filter field

A new `search` field is introduced for `REQ` messages from clients:

```
{
  // other fields on filter object...
  "search": <string>
}
```

`search` field is a string describing a query in a human-readable form, i.e. “best nostr apps”. Relays SHOULD interpret the query to the best of their ability and return events that match it. Relays SHOULD perform matching against `content` event field, and MAY perform matching against other fields if that makes sense in the context of a specific kind.

Results SHOULD be returned in descending order by quality of search result (as defined by the implementation), not by the usual `.created_at`. The `limit` filter SHOULD be applied after sorting by matching score. A query string may contain `key:value` pairs (two words separated by colon), these are extensions, relays SHOULD ignore extensions they don’t support.

Clients may specify several search filters, i.e. `["REQ", "", { "search": "orange" }, { "kinds": [1, 2], "search": "purple" }]`. Clients may include `kinds`, `ids` and other filter field to restrict the search results to particular event kinds.

Clients SHOULD use the supported_nips field to learn if a relay supports `search` filter. Clients MAY send `search` filter queries to any relay, if they are prepared to filter out extraneous responses from relays that do not support this NIP.

Clients SHOULD query several relays supporting this NIP to compensate for potentially different implementation details between relays.

Clients MAY verify that events returned by a relay match the specified query in a way that suits the client’s use case, and MAY stop querying relays that have low precision.

Relays SHOULD exclude spam from search results by default if they support some form of spam filtering.

### Extensions

Relay MAY support these extensions:

- `include:spam` - turn off spam filtering, if it was enabled by default
- `domain:<domain>` - include only events from users whose valid nip05 domain matches the domain
- `language:<two letter ISO 639-1 language code>` - include only events of a specified language
- `sentiment:<negative/neutral/positive>` - include only events of a specific sentiment
- `nsfw:<true/false>` - include or exclude nsfw events (default: true)

## NIP-51

## Lists

`draft` `optional`

This NIP defines lists of things that users can create. Lists can contain references to anything, and these references can be **public** or **private**.

Public items in a list are specified in the event `tags` array, while private items are specified in a JSON array that mimics the structure of the event `tags` array, but stringified and encrypted using the same scheme from NIP-04 (the shared key is computed using the author’s public and private key) and stored in the `.content`.

When new items are added to an existing list, clients SHOULD append them to the end of the list, so they are stored in chronological order.

### Types of lists

#### Standard lists

Standard lists use normal replaceable events, meaning users may only have a single list of each kind. They have special meaning and clients may rely on them to augment a user’s profile or browsing experience.

For example, *mute list* can contain the public keys of spammers and bad actors users don’t want to see in their feeds or receive annoying notifications from.

| name | kind | description | expected tag items |
|---|---|---|---|
| Mute list | 10000 | things the user doesn't want to see in their feeds | `"p"` (pubkeys), `"t"` (hashtags), `"word"` (lowercase string), `"e"` (threads) |
| Pinned notes | 10001 | events the user intends to showcase in their profile page | `"e"` (kind:1 notes) |
| Bookmarks | 10003 | uncategorized, "global" list of things a user wants to save | `"e"` (kind:1 notes), `"a"` (kind:30023 articles), `"t"` (hashtags), `"r"` (URLs) |
| Communities | 10004 | NIP-72 communities the user belongs to | `"a"` (kind:34550 community definitions) |
| Public chats | 10005 | NIP-28 chat channels the user is in | `"e"` (kind:40 channel definitions) |
| Blocked relays | 10006 | relays clients should never connect to | `"relay"` (relay URLs) |
| Search relays | 10007 | relays clients should use when performing search queries | `"relay"` (relay URLs) |
| Simple groups | 10009 | NIP-29 groups the user is in | `"group"` (NIP-29 group id + relay URL + optional group name), `"r"` for each relay in use |
| Interests | 10015 | topics a user may be interested in and pointers | `"t"` (hashtags) and `"a"` (kind:30015 interest set) |
| Emojis | 10030 | user preferred emojis and pointers to emoji sets | `"emoji"` (see NIP-30 ) and `"a"` (kind:30030 emoji set) |
| DM relays | 10050 | Where to receive NIP-17 direct messages | `"relay"` (see NIP-17 ) |
| Good wiki authors | 10101 | NIP-54 user recommended wiki authors | `"p"` (pubkeys) |
| Good wiki relays | 10102 | NIP-54 relays deemed to only host useful articles | `"relay"` (relay URLs) |

#### Sets

Sets are lists with well-defined meaning that can enhance the functionality and the UI of clients that rely on them. Unlike standard lists, users are expected to have more than one set of each kind, therefore each of them must be assigned a different `"d"` identifier.

For example, *relay sets* can be displayed in a dropdown UI to give users the option to switch to which relays they will publish an event or from which relays they will read the replies to an event; *curation sets* can be used by apps to showcase curations made by others tagged to different topics.

Aside from their main identifier, the `"d"` tag, sets can optionally have a `"title"`, an `"image"` and a `"description"` tags that can be used to enhance their UI.

| name | kind | description | expected tag items |
|---|---|---|---|
| Follow sets | 30000 | categorized groups of users a client may choose to check out in different circumstances | `"p"` (pubkeys) |
| Relay sets | 30002 | user-defined relay groups the user can easily pick and choose from during various operations | `"relay"` (relay URLs) |
| Bookmark sets | 30003 | user-defined bookmarks categories , for when bookmarks must be in labeled separate groups | `"e"` (kind:1 notes), `"a"` (kind:30023 articles), `"t"` (hashtags), `"r"` (URLs) |
| Curation sets | 30004 | groups of articles picked by users as interesting and/or belonging to the same category | `"a"` (kind:30023 articles), `"e"` (kind:1 notes) |
| Curation sets | 30005 | groups of videos picked by users as interesting and/or belonging to the same category | `"a"` (kind:34235 videos) |
| Kind mute sets | 30007 | mute pubkeys by kinds `"d"` tag MUST be the kind string | `"p"` (pubkeys) |
| Interest sets | 30015 | interest topics represented by a bunch of "hashtags" | `"t"` (hashtags) |
| Emoji sets | 30030 | categorized emoji groups | `"emoji"` (see NIP-30 ) |
| Release artifact sets | 30063 | groups of files of a software release | `"e"` (kind:1063 file metadata events), `"i"` (application identifier, typically reverse domain notation), `"version"` |

#### Deprecated standard lists

Some clients have used these lists in the past, but they should work on transitioning to the standard formats above.

| kind | "d" tag | use instead |
|---|---|---|
| 30000 | `"mute"` | kind 10000 *mute list* |
| 30001 | `"pin"` | kind 10001 *pin list* |

| kind | “d” tag | use instead |
| --- | --- | --- |
| 30001 | `"bookmark"` | kind 10003 *bookmarks list* |
| 30001 | `"communities"` | kind 10004 *communities list* |

### Examples

**A *mute list* with some public items and some encrypted items**

```
{
  "id": "a92a316b75e44cfdc19986c634049158d4206fcc0b7b9c7ccbcdabe28beebcd0",
  "pubkey": "854043ae8f1f97430ca8c1f1a090bdde6488bd5115c7a45307a2a212750ae4cb",
  "created_at": 1699597889,
  "kind": 10000,
  "tags": [
    ["p", "07caba282f76441955b695551c3c5c742e5b9202a3784780f8086fdcdc1da3a9"],
    ["p", "a55c15f5e41d5aebd236eca5e0142789c5385703f1a7485aa4b38d94fd18dcc4"]
  ],
  "content":
"TJob1dQrf2ndsmdbeGU+05HT5GMnBSx3fx8QdDY/g3NvCa7klfzgaQCmRZuo1d3WQjHDOjzSY1+MgTK5WjewFFumCcOZniWtOMSga
iv=S3rFeFr1gsYqmQA7bNnNTQ==",
  "sig":
"1173822c53261f8cffe7efbf43ba4a97a9198b3e402c2a1df130f42a8985a2d0d3430f4de350db184141e45ca844ab4e5364e

}
```

**A *curation set* of articles and notes about yaks**

```
{
  "id": "567b41fc9060c758c4216fe5f8d3df7c57daad7ae757fa4606f0c39d4dd220ef",
  "pubkey": "d6dc95542e18b8b7aec2f14610f55c335abebec76f3db9e58c254661d0593a0c",
  "created_at": 1695327657,
  "kind": 30004,
  "tags": [
    ["d", "jvdy9i4"],
    ["name", "Yaks"],
    ["picture", "https://cdn.britannica.com/40/188540-050-9AC748DE/Yak-Himalayas-
Nepal.jpg"],
    ["about", "The domestic yak, also known as the Tartary ox, grunting ox, or hairy
cattle, is a species of long-haired domesticated cattle found throughout the
Himalayan region of the Indian subcontinent, the Tibetan Plateau, Gilgit-Baltistan,
Tajikistan and as far north as Mongolia and Siberia."],
    ["a",
"30023:26dc95542e18b8b7aec2f14610f55c335abebec76f3db9e58c254661d0593a0c:95ODQzw3ajNoZ8SyMDOzQ"],

    ["a", "30023:54af95542e18b8b7aec2f14610f55c335abebec76f3db9e58c254661d0593a0c:1-
MYP8dAhramH9J5gJWKx"],
    ["a",
"30023:f8fe95542e18b8b7aec2f14610f55c335abebec76f3db9e58c254661d0593a0c:D2Tbd38bGrFvU0bIbvSMt"],

    ["e", "d78ba0d5dce22bfff9db0a9e996c9ef27e2c91051de0c4e1da340e0326b4941e"]
  ],
  "content": "",
  "sig":
"a9a4e2192eede77e6c9d24ddfab95ba3ff7c03fbd07ad011fff245abea431fb4d3787c2d04aad001cb039cb8de91d83ce30e9

}
```

**A *release artifact set* of an Example App**

```
{
  "id": "567b41fc9060c758c4216fe5f8d3df7c57daad7ae757fa4606f0c39d4dd220ef",
  "pubkey": "d6dc95542e18b8b7aec2f14610f55c335abebec76f3db9e58c254661d0593a0c",
  "created_at": 1695327657,
  "kind": 30063,
  "tags": [
    ["d", "ak8dy3v7"],
    ["i", "com.example.app"],
```

```
    ["version", "0.0.1"],
    ["title", "Example App"],
    ["image", "http://cdn.site/p/com.example.app/icon.png"],
    ["e", "d78ba0d5dce22bfff9db0a9e996c9ef27e2c91051de0c4e1da340e0326b4941e"], //
Windows exe
    ["e", "f27e2c91051de0c4e1da0d5dce22bfff9db0a9340e0326b4941ed78bae996c9e"], //
MacOS dmg
    ["e", "9d24ddfab95ba3ff7c03fbd07ad011fff245abea431fb4d3787c2d04aad02332"], //
Linux AppImage
    ["e", "340e0326b340e0326b4941ed78ba340e0326b4941ed78ba340e0326b49ed78ba"] // PWA
  ],
  "content": "Example App is a decentralized marketplace for apps",
  "sig":
"a9a4e2192eede77e6c9d24ddfab95ba3ff7c03fbd07ad011fff245abea431fb4d3787c2d04aad001cb039cb8de91d83ce30e9

}
```

### Encryption process pseudocode

```
val private_items = [
  ["p", "07caba282f76441955b695551c3c5c742e5b9202a3784780f8086fdcdc1da3a9"],
  ["a", "a55c15f5e41d5aebd236eca5e0142789c5385703f1a7485aa4b38d94fd18dcc4"],
]
val base64blob = nip04.encrypt(json.encode_to_string(private_items))
event.content = base64blob
```

## NIP-52

## Calendar Events

`draft` `optional`

This specification defines calendar events representing an occurrence at a specific moment or between moments. These calendar events are *addressable* and deletable per NIP-09 .

Unlike the term `calendar event` specific to this NIP, the term `event` is used broadly in all the NIPs to describe any Nostr event. The distinction is being made here to discern between the two terms.

### Calendar Events

There are two types of calendar events represented by different kinds: date-based and time-based calendar events. Calendar events are not required to be part of a calendar .

#### Date-Based Calendar Event

This kind of calendar event starts on a date and ends before a different date in the future. Its use is appropriate for all-day or multi-day events where time and time zone hold no significance. e.g., anniversary, public holidays, vacation days.

##### Format

The format uses an *addressable event* of `kind:31922`.

The `.content` of these events should be a detailed description of the calendar event. It is required but can be an empty string.

The list of tags are as follows:

- `d` (required) universally unique identifier (UUID). Generated by the client creating the calendar event.
- `title` (required) title of the calendar event
- `start` (required) inclusive start date in ISO 8601 format (YYYY-MM-DD). Must be less than `end`, if it exists.
- `end` (optional) exclusive end date in ISO 8601 format (YYYY-MM-DD). If omitted, the calendar event ends on the same date as `start`.
- `location` (optional, repeated) location of the calendar event. e.g. address, GPS coordinates, meeting room name, link to video call
- `g` (optional) geohash to associate calendar event with a searchable physical location
- `p` (optional, repeated) 32-bytes hex pubkey of a participant, optional recommended relay URL, and participant's role in the meeting

- `t` (optional, repeated) hashtag to categorize calendar event
- `r` (optional, repeated) references / links to web pages, documents, video calls, recorded videos, etc.

The following tags are deprecated:

- `name` name of the calendar event. Use only if `title` is not available.

```
{
  "id": <32-bytes lowercase hex-encoded SHA-256 of the the serialized event data>,
  "pubkey": <32-bytes lowercase hex-encoded public key of the event creator>,
  "created_at": <Unix timestamp in seconds>,
  "kind": 31922,
  "content": "<description of calendar event>",
  "tags": [
    ["d", "<UUID>"],

    ["title", "<title of calendar event>"],

    // Dates
    ["start", "<YYYY-MM-DD>"],
    ["end", "<YYYY-MM-DD>"],

    // Location
    ["location", "<location>"],
    ["g", "<geohash>"],

    // Participants
    ["p", "<32-bytes hex of a pubkey>", "<optional recommended relay URL>", "
<role>"],
    ["p", "<32-bytes hex of a pubkey>", "<optional recommended relay URL>", "
<role>"],

    // Hashtags
    ["t", "<tag>"],
    ["t", "<tag>"],

    // Reference links
    ["r", "<url>"],
    ["r", "<url>"]
  ]
}
```

**Time-Based Calendar Event**

This kind of calendar event spans between a start time and end time.

##### Format

The format uses an *addressable event* kind `31923`.

The `.content` of these events should be a detailed description of the calendar event. It is required but can be an empty string.

The list of tags are as follows:

- `d` (required) universally unique identifier (UUID). Generated by the client creating the calendar event.
- `title` (required) title of the calendar event
- `start` (required) inclusive start Unix timestamp in seconds. Must be less than `end`, if it exists.
- `end` (optional) exclusive end Unix timestamp in seconds. If omitted, the calendar event ends instantaneously.
- `start_tzid` (optional) time zone of the start timestamp, as defined by the IANA Time Zone Database. e.g., `America/Costa_Rica`
- `end_tzid` (optional) time zone of the end timestamp, as defined by the IANA Time Zone Database. e.g., `America/Costa_Rica`. If omitted and `start_tzid` is provided, the time zone of the end timestamp is the same as the start timestamp.
- `summary` (optional) brief description of the calendar event
- `image` (optional) url of an image to use for the event

- `location` (optional, repeated) location of the calendar event. e.g. address, GPS coordinates, meeting room name, link to video call
- `g` (optional) geohash to associate calendar event with a searchable physical location
- `p` (optional, repeated) 32-bytes hex pubkey of a participant, optional recommended relay URL, and participant's role in the meeting
- `l` (optional, repeated) label to categorize calendar event. e.g. `audiospace` to denote a scheduled event from a live audio space implementation such as cornychat.com
- `t` (optional, repeated) hashtag to categorize calendar event
- `r` (optional, repeated) references / links to web pages, documents, video calls, recorded videos, etc.

The following tags are deprecated:

- `name` name of the calendar event. Use only if `title` is not available.

```
{
  "id": <32-bytes lowercase hex-encoded SHA-256 of the the serialized event data>,
  "pubkey": <32-bytes lowercase hex-encoded public key of the event creator>,
  "created_at": <Unix timestamp in seconds>,
  "kind": 31923,
  "content": "<description of calendar event>",
  "tags": [
    ["d", "<UUID>"],

    ["title", "<title of calendar event>"],
    ["summary", "<brief description of the calendar event>"],
    ["image", "<string with image URI>"],

    // Timestamps
    ["start", "<Unix timestamp in seconds>"],
    ["end", "<Unix timestamp in seconds>"],

    ["start_tzid", "<IANA Time Zone Database identifier>"],
    ["end_tzid", "<IANA Time Zone Database identifier>"],

    // Location
    ["location", "<location>"],
    ["g", "<geohash>"],

    // Participants
    ["p", "<32-bytes hex of a pubkey>", "<optional recommended relay URL>", "
<role>"],
    ["p", "<32-bytes hex of a pubkey>", "<optional recommended relay URL>", "
<role>"],

    // Labels (example using com.cornychat namespace denoting the event as an
audiospace)
    ["L", "com.cornychat"],
    ["l", "audiospace", "com.cornychat"],

    // Hashtags
    ["t", "<tag>"],
    ["t", "<tag>"],

    // Reference links
    ["r", "<url>"],
    ["r", "<url>"]
  ]
}
```

## Calendar

A calendar is a collection of calendar events, represented as a custom replaceable list event using kind `31924`. A user can have multiple calendars. One may create a calendar to segment calendar events for specific purposes. e.g., personal, work, travel, meetups, and conferences.

#### Format

The `.content` of these events should be a detailed description of the calendar. It is required but can be an empty string.

The format uses a custom replaceable list of kind `31924` with a list of tags as described below:

- `d` (required) universally unique identifier. Generated by the client creating the calendar.
- `title` (required) calendar title
- `a` (repeated) reference tag to kind `31922` or `31923` calendar event being responded to

```
{
  "id": <32-bytes lowercase hex-encoded SHA-256 of the the serialized event data>,
  "pubkey": <32-bytes lowercase hex-encoded public key of the event creator>,
  "created_at": <Unix timestamp in seconds>,
  "kind": 31924,
  "content": "<description of calendar>",
  "tags": [
    ["d", "<UUID>"],
    ["title", "<calendar title>"],
    ["a", "<31922 or 31923>:<calendar event author pubkey>:<d-identifier of calendar
event>", "<optional relay url>"],
    ["a", "<31922 or 31923>:<calendar event author pubkey>:<d-identifier of calendar
event>", "<optional relay url>"]
  ]
}
```

### Calendar Event RSVP

A calendar event RSVP is a response to a calendar event to indicate a user's attendance intention.

If a calendar event tags a pubkey, that can be interpreted as the calendar event creator inviting that user to attend. Clients MAY choose to prompt the user to RSVP for the calendar event.

Any user may RSVP, even if they were not tagged on the calendar event. Clients MAY choose to prompt the calendar event creator to invite the user who RSVP'd. Clients also MAY choose to ignore these RSVPs.

This NIP is intentionally not defining who is authorized to attend a calendar event if the user who RSVP'd has not been tagged. It is up to the calendar event creator to determine the semantics.

This NIP is also intentionally not defining what happens if a calendar event changes after an RSVP is submitted.

The RSVP MUST have an `a` tag of the event coordinates to the calendar event, and optionally an `e` tag of the id of the specific calendar event revision. If an `e` tag is present, clients SHOULD interpret it as an indication that the RSVP is a response to that revision of the calendar event, and MAY interpret it to not necessarily apply to other revisions of the calendar event.

The RSVP MAY tag the author of the calendar event it is in response to using a `p` tag so that clients can easily query all RSVPs that pertain to the author.

#### Format

The format uses an *addressable event* kind `31925`.

The `.content` of these events is optional and should be a free-form note that adds more context to this calendar event response.

The list of tags are as follows:

- `a` (required) coordinates to a kind `31922` or `31923` calendar event being responded to.
- `e` (optional) event id of a kind `31922` or `31923` calendar event being responded to.
- `d` (required) universally unique identifier. Generated by the client creating the calendar event RSVP.
- `status` (required) `accepted`, `declined`, or `tentative`. Determines attendance status to the referenced calendar event.
- `fb` (optional) `free` or `busy`. Determines if the user would be free or busy for the duration of the calendar event. This tag must be omitted or ignored if the `status` label is set to `declined`.
- `p` (optional) pubkey of the author of the calendar event being responded to.

```
{
  "id": <32-bytes lowercase hex-encoded SHA-256 of the the serialized event data>,
  "pubkey": <32-bytes lowercase hex-encoded public key of the event creator>,
  "created_at": <Unix timestamp in seconds>,
```

```
  "kind": 31925,
  "content": "<note>",
  "tags": [
    ["e", "<kind 31922 or 31923 event id", "<optional recommended relay URL>"]
    ["a", "<31922 or 31923>:<calendar event author pubkey>:<d-identifier of calendar
event>", "<optional recommended relay URL>"],
    ["d", "<UUID>"],
    ["status", "<accepted/declined/tentative>"],
    ["fb", "<free/busy>"],
    ["p", "<hex pubkey of kind 31922 or 31923 event>", "<optional recommended relay
URL>"]
  ]
}
```

### Unsolved Limitations

- No private events

### Intentionally Unsupported Scenarios

#### Recurring Calendar Events

Recurring calendar events come with a lot of complexity, making it difficult for software and humans to deal with. This complexity includes time zone differences between invitees, daylight savings, leap years, multiple calendar systems, one-off changes in schedule or other metadata, etc.

This NIP intentionally omits support for recurring calendar events and pushes that complexity up to clients to manually implement if they desire. i.e., individual calendar events with duplicated metadata represent recurring calendar events.

## NIP-53

## Live Activities

`draft` `optional`

Service providers want to offer live activities to the Nostr network in such a way that participants can easily log and query by clients. This NIP describes a general framework to advertise the involvement of pubkeys in such live activities.

### Concepts

#### Live Event

A special event with `kind:30311` "Live Event" is defined as an *addressable event* of public `p` tags. Each `p` tag SHOULD have a **displayable** marker name for the current role (e.g. `Host`, `Speaker`, `Participant`) of the user in the event and the relay information MAY be empty. This event will be constantly updated as participants join and leave the activity.

For example:

```
{
  "kind": 30311,
  "tags": [
    ["d", "<unique identifier>"],
    ["title", "<name of the event>"],
    ["summary", "<description>"],
    ["image", "<preview image url>"],
    ["t", "hashtag"]
    ["streaming", "<url>"],
    ["recording", "<url>"], // used to place the edited video once the activity is
over
    ["starts", "<unix timestamp in seconds>"],
    ["ends", "<unix timestamp in seconds>"],
    ["status", "<planned, live, ended>"],
    ["current_participants", "<number>"],
    ["total_participants", "<number>"],
    ["p", "91cf9..4e5ca", "wss://provider1.com/", "Host", "<proof>"],
    ["p", "14aeb..8dad4", "wss://provider2.com/nostr", "Speaker"],
```

```
    ["p", "612ae..e610f", "ws://provider3.com/ws", "Participant"],
    ["relays", "wss://one.com", "wss://two.com", /*...*/]
  ],
  "content": "",
  // other fields...
}
```

A distinct `d` tag should be used for each activity. All other tags are optional.

Providers SHOULD keep the participant list small (e.g. under 1000 users) and, when limits are reached, Providers SHOULD select which participants get named in the event. Clients should not expect a comprehensive list. Once the activity ends, the event can be deleted or updated to summarize the activity and provide async content (e.g. recording of the event).

Clients are expected to subscribe to `kind:30311` events in general or for given follow lists and statuses. Clients MAY display participants' roles in activities as well as access points to join the activity.

Live Activity management clients are expected to constantly update `kind:30311` during the event. Clients MAY choose to consider `status=live` events after 1hr without any update as `ended`. The `starts` and `ends` timestamp SHOULD be updated when the status changes to and from `live`

The activity MUST be linked to using the NIP-19 `naddr` code along with the `a` tag.

#### Proof of Agreement to Participate

Event owners can add proof as the 5th term in each `p` tag to clarify the participant's agreement in joining the event. The proof is a signed SHA256 of the complete `a` Tag of the event (`kind:pubkey:dTag`) by each `p`'s private key, encoded in hex.

Clients MAY only display participants if the proof is available or MAY display participants as "invited" if the proof is not available.

This feature is important to avoid malicious event owners adding large account holders to the event, without their knowledge, to lure their followers into the malicious owner's trap.

#### Live Chat Message

Event `kind:1311` is live chat's channel message. Clients MUST include the `a` tag of the activity with a `root` marker. Other Kind-1 tags such as `reply` and `mention` can also be used.

```
{
  "kind": 1311,
  "tags": [
    ["a", "30311:<Community event author pubkey>:<d-identifier of the community>", "
<Optional relay url>", "root"],
  ],
  "content": "Zaps to live streams is beautiful.",
  // other fields...
}
```

### Use Cases

Common use cases include meeting rooms/workshops, watch-together activities, or event spaces, such as zap.stream .

### Example

#### Live Streaming

```
{
  "id": "57f28dbc264990e2c61e80a883862f7c114019804208b14da0bff81371e484d2",
  "pubkey": "1597246ac22f7d1375041054f2a4986bd971d8d196d7997e48973263ac9879ec",
  "created_at": 1687182672,
  "kind": 30311,
  "tags": [
    ["d", "demo-cf-stream"],
    ["title", "Adult Swim Metalocalypse"],
    ["summary", "Live stream from IPTV-ORG collection"],
    ["streaming", "https://adultswim-
```

```
vodlive.cdn.turner.com/live/metalocalypse/stream.m3u8"],
    ["starts", "1687182672"],
    ["status", "live"],
    ["t", "animation"],
    ["t", "iptv"],
    ["image", "https://i.imgur.com/CaKq6Mt.png"]
  ],
  "content": "",
  "sig":
"5bc7a60f5688effa5287244a24768cbe0dcd854436090abc3bef172f7f5db1410af4277508dbafc4f70a754a891c90ce3b966
}
```


**Live Streaming chat message**


```
{
  "id": "97aa81798ee6c5637f7b21a411f89e10244e195aa91cb341bf49f718e36c8188",
  "pubkey": "3f770d65d3a764a9c5cb503ae123e62ec7598ad035d836e2a810f3877a745b24",
  "created_at": 1687286726,
  "kind": 1311,
  "tags": [
    ["a",
"30311:1597246ac22f7d1375041054f2a4986bd971d8d196d7997e48973263ac9879ec:demo-cf-
stream", "", "root"]
  ],
  "content": "Zaps to live streams is beautiful.",
  "sig":
"997f62ddfc0827c121043074d50cfce7a528e978c575722748629a4137c45b75bdbc84170bedc723ef0a5a4c3daebf1fef2e9
}
```


## NIP-54

## Wiki

`draft` `optional`

This NIP defines `kind:30818` (an *addressable event*) for descriptions (or encyclopedia entries) of particular subjects, and it's expected that multiple people will write articles about the exact same subjects, with either small variations or completely independent content.

Articles are identified by lowercase, normalized ascii `d` tags.

**Articles**


```
{
  "content": "A wiki is a hypertext publication collaboratively edited and managed
by its own audience.",
  "tags": [
    ["d", "wiki"],
    ["title", "Wiki"],
  ]
}
```


**`d` tag normalization rules**

- Any non-letter character MUST be converted to a `-`.
- All letters MUST be converted to lowercase.

**Content**

The `content` should be Asciidoc with two extra functionalities: **wikilinks** and **nostr:...** links.

Unlike normal Asciidoc links `http://example.com[]` that link to external webpages, wikilinks `[[]]` link to other articles in the wiki. In this case, the wiki is the entirety of Nostr. Clicking on a wikilink should cause the client to ask relays for events with `d` tags equal to the target of that wikilink.

Wikilinks can take these two forms:

1. `[[Target Page]]` – in this case it will link to the page `target-page` (according to `d` tag normalization rules above) and be displayed as `Target Page`;
2. `[[target page|see this]]` – in this case it will link to the page `target-page`, but will be displayed as `see this`.

`nostr:...` links, as per NIP-21 , should link to profiles or arbitrary Nostr events. Although it is not recommended to link to specific versions of articles – instead the *wikilink* syntax should be preferred, since it should be left to the reader and their client to decide what version of any given article they want to read.

#### Optional extra tags

- `title`: for when the display title should be different from the `d` tag.
- `summary`: for display in lists.
- `a` and `e`: for referencing the original event a wiki article was forked from.

#### Merge Requests

Event `kind:818` represents a request to merge from a forked article into the source. It is directed to a pubkey and references the original article and the modified event.

[INSERT EVENT EXAMPLE]

#### Redirects

Event `kind:30819` is also defined to stand for "wiki redirects", i.e. if one thinks `Shell structure` should redirect to `Thin-shell structure` they can issue one of these events instead of replicating the content. These events can be used for automatically redirecting between articles on a client, but also for generating crowdsourced "disambiguation" pages (common in Wikipedia ).

[INSERT EVENT EXAMPLE]

## How to decide what article to display

As there could be many articles for each given name, some kind of prioritization must be done by clients. Criteria for this should vary between users and clients, but some means that can be used are described below:

#### Reactions

NIP-25 reactions are very simple and can be used to create a simple web-of-trust between wiki article writers and their content. While just counting a raw number of "likes" is unproductive, reacting to any wiki article event with a `+` can be interpreted as a recommendation for that article specifically and a partial recommendation of the author of that article. When 2 or 3-level deep recommendations are followed, suddenly a big part of all the articles may have some form of tagging.

#### Relays

NIP-51 lists of relays can be created with the kind 10102 and then used by wiki clients in order to determine where to query articles first and to rank these differently in relation to other events fetched from other relays.

#### Contact lists

NIP-02 contact lists can form the basis of a recommendation system that is then expanded with relay lists and reaction lists through nested queries. These lists form a good starting point only because they are so widespread.

#### Wiki-related contact lists

NIP-51 lists can also be used to create a list of users that are trusted only in the context of wiki authorship or wiki curationship.

## Forks

Wiki-events can tag other wiki-events with a `fork` marker to specify that this event came from a different version. Both `a` and `e` tags SHOULD be used and have the `fork` marker applied, to identify the exact version it was forked from.

## Deference

Wiki-events can tag other wiki-events with a `defer` marker to indicate that it considers someone else's entry as a "better" version of itself. If using a `defer` marker both `a` and `e` tags SHOULD be used.

This is a stronger signal of trust than a `+` reaction.

This marker is useful when a user edits someone else's entry; if the original author includes the editor's changes and the editor doesn't want to keep/maintain an independent version, the `link` tag could effectively be a considered a "deletion" of the editor's version and putting that pubkey's WoT weight behind the original author's version.

## Why Asciidoc?

Wikitext is garbage and Markdown is not powerful enough (besides being too freeform and unspecified and prone to generate incompatibilities in the future).

Asciidoc has a strict spec, multiple implementations in many languages, and support for features that are very much necessary in a wiki article, like *sidebars*, *tables* (with rich markup inside cells), many levels of *headings*, *footnotes*, *superscript* and *subscript* markup and *description lists*. It is also arguably easier to read in its plaintext format than Markdown (and certainly much better than Wikitext).

## Appendix 1: Merge requests

Users can request other users to get their entries merged into someone else's entry by creating a `kind:818` event.

```
{
  "content": "I added information about how to make hot ice-creams",
  "kind": 818,
  "tags": [
    [ "a", "30818:<destination-pubkey>:hot-ice-creams", "<relay-url>" ],
    [ "e", "<version-against-which-the-modification-was-made>", "<relay-url>" ],
    [ "p", "<destination-pubkey>" ],
    [ "e", "<version-to-be-merged>", "<relay-url>", "source" ]
  ]
}
```

`.content`: an optional explanation detailing why this merge is being requested. `a` tag: tag of the article which should be modified (i.e. the target of this merge request). `e` tag: optional version of the article in which this modifications is based `e` tag with `source` marker: the ID of the event that should be merged. This event id MUST be of a `kind:30818` as defined in this NIP.

The destination-pubkey is the pubkey being requested to merge something into their article can create [[NIP-25]] reactions that tag the `kind:818` event with `+` or `-`

## NIP-55

## Android Signer Application

`draft` `optional`

This NIP describes a method for 2-way communication between an Android signer and any Nostr client on Android. The Android signer is an Android Application and the client can be a web client or an Android application.

## Usage for Android applications

The Android signer uses Intents and Content Resolvers to communicate between applications.

To be able to use the Android signer in your application you should add this to your AndroidManifest.xml:

```
<queries>
  <intent>
    <action android:name="android.intent.action.VIEW" />
    <category android:name="android.intent.category.BROWSABLE" />
    <data android:scheme="nostrsigner" />
  </intent>
</queries>
```

Then you can use this function to check if there's a signer application installed:

```
fun isExternalSignerInstalled(context: Context): Boolean {
  val intent =
```

```
    Intent().apply {
      action = Intent.ACTION_VIEW
      data = Uri.parse("nostrsigner:")
    }
  val infos = context.packageManager.queryIntentActivities(intent, 0)
  return infos.size > 0
}
```

### Using Intents

To get the result back from the Signer Application you should use `registerForActivityResult` or `rememberLauncherForActivityResult` in Kotlin. If you are using another framework check the documentation of your framework or a third party library to get the result.

```
val launcher = rememberLauncherForActivityResult(
    contract = ActivityResultContracts.StartActivityForResult(),
    onResult = { result ->
        if (result.resultCode != Activity.RESULT_OK) {
            Toast.makeText(
                context,
                "Sign request rejected",
                Toast.LENGTH_SHORT
            ).show()
        } else {
            val result = activityResult.data?.getStringExtra("result")
            // Do something with result ...
        }
    }
)
```

Create the Intent using the **nostrsigner** scheme:

```
val intent = Intent(Intent.ACTION_VIEW, Uri.parse("nostrsigner:$content"))
```

Set the Signer package name:

```
intent.`package` = "com.example.signer"
```

If you are sending multiple intents without awaiting you can add some intent flags to sign all events without opening multiple times the signer

```
intent.addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP or Intent.FLAG_ACTIVITY_CLEAR_TOP)
```

If you are developing a signer application them you need to add this to your AndroidManifest.xml so clients can use the intent flags above

```
android:launchMode="singleTop"
```

Signer MUST answer multiple permissions with an array of results

```
val results = listOf(
  Result(
    package = signerPackageName,
    result = eventSignture,
    id = intentId
  )
)

val json = results.toJson()

intent.putExtra("results", json)
```

Send the Intent:

```
launcher.launch(intent)
```

**Methods**

- **get_public_key**
  - params:

```
val intent = Intent(Intent.ACTION_VIEW, Uri.parse("nostrsigner:"))
intent.`package` = "com.example.signer"
intent.putExtra("type", "get_public_key")
// You can send some default permissions for the user to authorize for ever
val permissions = listOf(
  Permission(
      type = "sign_event",
      kind = 22242
  ),
  Permission(
      type = "nip44_decrypt"
  )
)
intent.putExtra("permissions", permissions.toJson())
context.startActivity(intent)
```

  - result:
    - If the user approved intent it will return the **pubkey** in the result field

```
val pubkey = intent.data?.getStringExtra("result")
// The package name of the signer application
val packageName = intent.data?.getStringExtra("package")
```

- **sign_event**
  - params:

```
val intent = Intent(Intent.ACTION_VIEW,
Uri.parse("nostrsigner:$eventJson"))
intent.`package` = "com.example.signer"
intent.putExtra("type", "sign_event")
// To handle results when not waiting between intents
intent.putExtra("id", event.id)
// Send the current logged in user pubkey
intent.putExtra("current_user", pubkey)

context.startActivity(intent)
```

  - result:
    - If the user approved intent it will return the **result**, **id** and **event** fields

```
val signature = intent.data?.getStringExtra("result")
// The id you sent
val id = intent.data?.getStringExtra("id")
val signedEventJson = intent.data?.getStringExtra("event")
```

- **nip04_encrypt**
  - params:

```
val intent = Intent(Intent.ACTION_VIEW,
Uri.parse("nostrsigner:$plaintext"))
intent.`package` = "com.example.signer"
intent.putExtra("type", "nip04_encrypt")
// to control the result in your application in case you are not waiting
the result before sending another intent
intent.putExtra("id", "some_id")
// Send the current logged in user pubkey
intent.putExtra("current_user", account.keyPair.pubkey)
// Send the hex pubkey that will be used for encrypting the data
intent.putExtra("pubkey", pubkey)

context.startActivity(intent)
```

- result:
    - If the user approved intent it will return the **result** and **id** fields

```
val encryptedText = intent.data?.getStringExtra("result")
// the id you sent
val id = intent.data?.getStringExtra("id")
```

- **nip44_encrypt**
    - params:

```
val intent = Intent(Intent.ACTION_VIEW,
Uri.parse("nostrsigner:$plaintext"))
intent.`package` = "com.example.signer"
intent.putExtra("type", "nip44_encrypt")
// to control the result in your application in case you are not waiting
the result before sending another intent
intent.putExtra("id", "some_id")
// Send the current logged in user pubkey
intent.putExtra("current_user", account.keyPair.pubkey)
// Send the hex pubkey that will be used for encrypting the data
intent.putExtra("pubkey", pubkey)

context.startActivity(intent)
```

    - result:
        - If the user approved intent it will return the **signature** and **id** fields

```
val encryptedText = intent.data?.getStringExtra("signature")
// the id you sent
val id = intent.data?.getStringExtra("id")
```

- **nip04_decrypt**
    - params:

```
val intent = Intent(Intent.ACTION_VIEW,
Uri.parse("nostrsigner:$encryptedText"))
intent.`package` = "com.example.signer"
intent.putExtra("type", "nip04_decrypt")
// to control the result in your application in case you are not waiting
the result before sending another intent
intent.putExtra("id", "some_id")
// Send the current logged in user pubkey
intent.putExtra("current_user", account.keyPair.pubkey)
// Send the hex pubkey that will be used for decrypting the data
intent.putExtra("pubkey", pubkey)

context.startActivity(intent)
```

    - result:
        - If the user approved intent it will return the **result** and **id** fields

```
val plainText = intent.data?.getStringExtra("result")
// the id you sent
val id = intent.data?.getStringExtra("id")
```

- **nip44_decrypt**
    - params:

```
val intent = Intent(Intent.ACTION_VIEW,
Uri.parse("nostrsigner:$encryptedText"))
intent.`package` = "com.example.signer"
intent.putExtra("type", "nip04_decrypt")
// to control the result in your application in case you are not waiting
the result before sending another intent
```

```
intent.putExtra("id", "some_id")
// Send the current logged in user pubkey
intent.putExtra("current_user", account.keyPair.pubkey)
// Send the hex pubkey that will be used for decrypting the data
intent.putExtra("pubkey", pubkey)

context.startActivity(intent)
```

- result:
  - If the user approved intent it will return the **result** and **id** fields

```
val plainText = intent.data?.getStringExtra("result")
// the id you sent
val id = intent.data?.getStringExtra("id")
```

- **get_relays**
  - params:

```
val intent = Intent(Intent.ACTION_VIEW, Uri.parse("nostrsigner:"))
intent.`package` = "com.example.signer"
intent.putExtra("type", "get_relays")
// to control the result in your application in case you are not waiting
the result before sending another intent
intent.putExtra("id", "some_id")
// Send the current logged in user pubkey
intent.putExtra("current_user", account.keyPair.pubkey)

context.startActivity(intent)
```

  - result:
    - If the user approved intent it will return the **result** and **id** fields

```
val relayJsonText = intent.data?.getStringExtra("result")
// the id you sent
val id = intent.data?.getStringExtra("id")
```

- **decrypt_zap_event**
  - params:

```
val intent = Intent(Intent.ACTION_VIEW,
Uri.parse("nostrsigner:$eventJson"))
intent.`package` = "com.example.signer"
intent.putExtra("type", "decrypt_zap_event")
// to control the result in your application in case you are not waiting
the result before sending another intent
intent.putExtra("id", "some_id")
// Send the current logged in user pubkey
intent.putExtra("current_user", account.keyPair.pubkey)
context.startActivity(intent)
```

  - result:
    - If the user approved intent it will return the **result** and **id** fields

```
val eventJson = intent.data?.getStringExtra("result")
// the id you sent
val id = intent.data?.getStringExtra("id")
```

### Using Content Resolver

To get the result back from Signer Application you should use contentResolver.query in Kotlin. If you are using another framework check the documentation of your framework or a third party library to get the result.

If the user did not check the “remember my choice” option, the pubkey is not in Signer Application or the signer type is not recognized the `contentResolver` will return null

For the SIGN_EVENT type Signer Application returns two columns “result” and “event”. The column event is the signed event json

For the other types Signer Application returns the column “result”

If the user chose to always reject the event, signer application will return the column “rejected” and you should not open signer application

**Methods**

- **get_public_key**
  - params:

```
val result = context.contentResolver.query(
    Uri.parse("content://com.example.signer.GET_PUBLIC_KEY"),
    listOf("login"),
    null,
    null,
    null
)
```

  - result:
    - Will return the **pubkey** in the result column

```
   if (result == null) return

   if (result.moveToFirst()) {
       val index = it.getColumnIndex("result")
       if (index < 0) return
       val pubkey = it.getString(index)
   }
```

- **sign_event**
  - params:

```
val result = context.contentResolver.query(
    Uri.parse("content://com.example.signer.SIGN_EVENT"),
    listOf("$eventJson", "", "${logged_in_user_pubkey}"),
    null,
    null,
    null
)
```

  - result:
    - Will return the **result** and the **event** columns

```
   if (result == null) return

   if (result.moveToFirst()) {
       val index = it.getColumnIndex("result")
       val indexJson = it.getColumnIndex("event")
       val signature = it.getString(index)
       val eventJson = it.getString(indexJson)
   }
```

- **nip04_encrypt**
  - params:

```
val result = context.contentResolver.query(
    Uri.parse("content://com.example.signer.NIP04_ENCRYPT"),
    listOf("$plainText", "${hex_pub_key}", "${logged_in_user_pubkey}"),
    null,
    null,
    null
)
```

- result:
    - Will return the **result** column

```
   if (result == null) return

   if (result.moveToFirst()) {
       val index = it.getColumnIndex("result")
       val encryptedText = it.getString(index)
   }
```

- **nip44_encrypt**
    - params:

```
val result = context.contentResolver.query(
    Uri.parse("content://com.example.signer.NIP44_ENCRYPT"),
    listOf("$plainText", "${hex_pub_key}", "${logged_in_user_pubkey}"),
    null,
    null,
    null
)
```

    - result:
        - Will return the **result** column

```
   if (result == null) return

   if (result.moveToFirst()) {
       val index = it.getColumnIndex("result")
       val encryptedText = it.getString(index)
   }
```

- **nip04_decrypt**
    - params:

```
val result = context.contentResolver.query(
    Uri.parse("content://com.example.signer.NIP04_DECRYPT"),
    listOf("$encryptedText", "${hex_pub_key}", "${logged_in_user_pubkey}"),
    null,
    null,
    null
)
```

    - result:
        - Will return the **result** column

```
   if (result == null) return

   if (result.moveToFirst()) {
       val index = it.getColumnIndex("result")
       val encryptedText = it.getString(index)
   }
```

- **nip44_decrypt**
    - params:

```
val result = context.contentResolver.query(
    Uri.parse("content://com.example.signer.NIP44_DECRYPT"),
    listOf("$encryptedText", "${hex_pub_key}", "${logged_in_user_pubkey}"),
    null,
    null,
    null
)
```

    - result:

        - Will return the **result** column

```
    if (result == null) return

    if (result.moveToFirst()) {
        val index = it.getColumnIndex("result")
        val encryptedText = it.getString(index)
    }
```

- **get_relays**
    - params:

```
val result = context.contentResolver.query(
    Uri.parse("content://com.example.signer.GET_RELAYS"),
    listOf("${logged_in_user_pubkey}"),
    null,
    null,
    null
)
```

    - result:
        - Will return the **result** column

```
    if (result == null) return

    if (result.moveToFirst()) {
        val index = it.getColumnIndex("result")
        val relayJsonText = it.getString(index)
    }
```

- **decrypt_zap_event**
    - params:

```
val result = context.contentResolver.query(
    Uri.parse("content://com.example.signer.DECRYPT_ZAP_EVENT"),
    listOf("$eventJson", "", "${logged_in_user_pubkey}"),
    null,
    null,
    null
)
```

    - result:
        - Will return the **result** column

```
    if (result == null) return

    if (result.moveToFirst()) {
        val index = it.getColumnIndex("result")
        val eventJson = it.getString(index)
    }
```

## Usage for Web Applications

Since web applications can't receive a result from the intent, you should add a modal to paste the signature or the event json or create a callback url.

If you send the callback url parameter, Signer Application will send the result to the url.

If you don't send a callback url, Signer Application will copy the result to the clipboard.

You can configure the `returnType` to be **signature** or **event**.

Android intents and browser urls have limitations, so if you are using the `returnType` of **event** consider using the parameter **compressionType=gzip** that will return "Signer1" + Base64 gzip encoded event json

### Methods

- **get_public_key**
  - params:

```
window.href = `nostrsigner:?
compressionType=none&returnType=signature&type=get_public_key&callbackUrl=https://example.com/
event=`;
```

- **sign_event**
  - params:

```
window.href = `nostrsigner:${eventJson}?
compressionType=none&returnType=signature&type=sign_event&callbackUrl=https://example.com/?
event=`;
```

- **nip04_encrypt**
  - params:

```
window.href = `nostrsigner:${plainText}?
pubkey=${hex_pub_key}&compressionType=none&returnType=signature&type=nip04_encrypt&callbackUrl
event=`;
```

- **nip44_encrypt**
  - params:

```
window.href = `nostrsigner:${plainText}?
pubkey=${hex_pub_key}&compressionType=none&returnType=signature&type=nip44_encrypt&callbackUrl
event=`;
```

- **nip04_decrypt**
  - params:

```
window.href = `nostrsigner:${encryptedText}?
pubkey=${hex_pub_key}&compressionType=none&returnType=signature&type=nip04_decrypt&callbackUrl
event=`;
```

- **nip44_decrypt**
  - params:

```
window.href = `nostrsigner:${encryptedText}?
pubkey=${hex_pub_key}&compressionType=none&returnType=signature&type=nip44_decrypt&callbackUrl
event=`;
```

- **get_relays**
  - params:

```
window.href = `nostrsigner:?
compressionType=none&returnType=signature&type=get_relays&callbackUrl=https://example.com/?
event=`;
```

- **decrypt_zap_event**
  - params:

```
window.href = `nostrsigner:${eventJson}?
compressionType=none&returnType=signature&type=decrypt_zap_event&callbackUrl=https://example.c
event=`;
```

### Example

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
    <title>Document</title>
</head>
<body>
    <h1>Test</h1>

    <script>
        window.onload = function() {
            var url = new URL(window.location.href);
            var params = url.searchParams;
            if (params) {
                var param1 = params.get("event");
                if (param1) alert(param1)
            }
            let json = {
                kind: 1,
                content: "test"
            }
            let encodedJson = encodeURIComponent(JSON.stringify(json))
            var newAnchor = document.createElement("a");
            newAnchor.href = `nostrsigner:${encodedJson}?
compressionType=none&returnType=signature&type=sign_event&callbackUrl=https://example.com/?
event=`;
            newAnchor.textContent = "Open External Signer";
            document.body.appendChild(newAnchor)
        }
    </script>
</body>
</html>
```

## NIP-56

## Reporting

`optional`

A report is a `kind 1984` event that signals to users and relays that some referenced content is objectionable. The definition of objectionable is obviously subjective and all agents on the network (users, apps, relays, etc.) may consume and take action on them as they see fit.

The `content` MAY contain additional information submitted by the entity reporting the content.

## Tags

The report event MUST include a `p` tag referencing the pubkey of the user you are reporting.

If reporting a note, an `e` tag MUST also be included referencing the note id.

A `report type` string MUST be included as the 3rd entry to the `e` or `p` tag being reported, which consists of the following report types:

- `nudity` - depictions of nudity, porn, etc.
- `malware` - virus, trojan horse, worm, robot, spyware, adware, back door, ransomware, rootkit, kidnapper, etc.
- `profanity` - profanity, hateful speech, etc.
- `illegal` - something which may be illegal in some jurisdiction
- `spam` - spam
- `impersonation` - someone pretending to be someone else
- `other` - for reports that don't fit in the above categories

Some report tags only make sense for profile reports, such as `impersonation`

`l` and `L` tags MAY be also be used as defined in NIP-32 to support further qualification and querying.

## Example events

```
{
  "kind": 1984,
  "tags": [
    ["p", <pubkey>, "nudity"],
```

```
    ["L", "social.nos.ontology"],
    ["l", "NS-nud", "social.nos.ontology"]
  ],
  "content": "",
  // other fields...
}
```

```
{
  "kind": 1984,
  "tags": [
    ["e", <eventId>, "illegal"],
    ["p", <pubkey>]
  ],
  "content": "He's insulting the king!",
  // other fields...
}
```

```
{
  "kind": 1984,
  "tags": [
    ["p", <impersonator pubkey>, "impersonation"]
  ],
  "content": "Profile is impersonating nostr:<victim bech32 pubkey>",
  // other fields...
}
```

## Client behavior

Clients can use reports from friends to make moderation decisions if they choose to. For instance, if 3+ of your friends report a profile for `nudity`, clients can have an option to automatically blur photos from said account.

## Relay behavior

It is not recommended that relays perform automatic moderation using reports, as they can be easily gamed. Admins could use reports from trusted moderators to takedown illegal or explicit content if the relay does not allow such things.

## NIP-57

## Lightning Zaps

`draft` `optional`

This NIP defines two new event types for recording lightning payments between users. `9734` is a `zap request`, representing a payer's request to a recipient's lightning wallet for an invoice. `9735` is a `zap receipt`, representing the confirmation by the recipient's lightning wallet that the invoice issued in response to a `zap request` has been paid.

Having lightning receipts on nostr allows clients to display lightning payments from entities on the network. These can be used for fun or for spam deterrence.

### Protocol flow

1. Client calculates a recipient's lnurl pay request url from the `zap` tag on the event being zapped (see Appendix G), or by decoding their lud06 or lud16 field on their profile according to the lnurl specifications . The client MUST send a GET request to this url and parse the response. If `allowsNostr` exists and it is `true`, and if `nostrPubkey` exists and is a valid BIP 340 public key in hex, the client should associate this information with the user, along with the response's `callback`, `minSendable`, and `maxSendable` values.
2. Clients may choose to display a lightning zap button on each post or on a user's profile. If the user's lnurl pay request endpoint supports nostr, the client SHOULD use this NIP to request a `zap receipt` rather than a normal lnurl invoice.
3. When a user (the "sender") indicates they want to send a zap to another user (the "recipient"), the client should create a `zap request` event as described in Appendix A of this NIP and sign it.
4. Instead of publishing the `zap request`, the `9734` event should instead be sent to the `callback` url received from the lnurl pay endpoint for the recipient using a GET request. See Appendix B for details and an example.
5. The recipient's lnurl server will receive this `zap request` and validate it. See Appendix C for details on how to properly configure an lnurl server to support zaps, and Appendix D for details on how to validate the `nostr`

query parameter.

6. If the `zap request` is valid, the server should fetch a description hash invoice where the description is this `zap request` note and this note only. No additional lnurl metadata is included in the description. This will be returned in the response according to LUD06 .
7. On receiving the invoice, the client MAY pay it or pass it to an app that can pay the invoice.
8. Once the invoice is paid, the recipient's lnurl server MUST generate a `zap receipt` as described in Appendix E, and publish it to the `relays` specified in the `zap request`.
9. Clients MAY fetch `zap receipt`s on posts and profiles, but MUST authorize their validity as described in Appendix F. If the `zap request` note contains a non-empty `content`, it may display a zap comment. Generally clients should show users the `zap request` note, and use the `zap receipt` to show "zap authorized by …" but this is optional.

## Reference and examples

#### Appendix A: Zap Request Event

A `zap request` is an event of kind `9734` that is *not* published to relays, but is instead sent to a recipient's lnurl pay `callback` url. This event's `content` MAY be an optional message to send along with the payment. The event MUST include the following tags:

- `relays` is a list of relays the recipient's wallet should publish its `zap receipt` to. Note that relays should not be nested in an additional list, but should be included as shown in the example below.
- `amount` is the amount in *millisats* the sender intends to pay, formatted as a string. This is recommended, but optional.
- `lnurl` is the lnurl pay url of the recipient, encoded using bech32 with the prefix `lnurl`. This is recommended, but optional.
- `p` is the hex-encoded pubkey of the recipient.

In addition, the event MAY include the following tags:

- `e` is an optional hex-encoded event id. Clients MUST include this if zapping an event rather than a person.
- `a` is an optional event coordinate that allows tipping addressable events such as NIP-23 long-form notes.

Example:

```
{
  "kind": 9734,
  "content": "Zap!",
  "tags": [
    ["relays", "wss://nostr-pub.wellorder.com", "wss://anotherrelay.example.com"],
    ["amount", "21000"],
    ["lnurl",
"lnurl1dp68gurn8ghj7um5v93kketj9ehx2amn9uh8wetvdskkkmn0wahz7mrww4excup0dajx2mrv92x9xp"],

    ["p", "04c915daefee38317fa734444acee390a8269fe5810b2241e5e6dd343dfbecc9"],
    ["e", "9ae37aa68f48645127299e9453eb5d908a0cbb6058ff340d528ed4d37c8994fb"]
  ],
  "pubkey": "97c70a44366a6535c145b333f973ea86dfdc2d7a99da618c40c64705ad98e322",
  "created_at": 1679673265,
  "id": "30efed56a035b2549fcaeec0bf2c1595f9a9b3bb4b1a38abaf8ee9041c4b7d93",
  "sig":
"f2cb581a84ed10e4dc84937bd98e27acac71ab057255f6aa8dfa561808c981fe8870f4a03c1e3666784d82a9c802d3704e174

}
```

#### Appendix B: Zap Request HTTP Request

A signed `zap request` event is not published, but is instead sent using a HTTP GET request to the recipient's `callback` url, which was provided by the recipient's lnurl pay endpoint. This request should have the following query parameters defined:

- `amount` is the amount in *millisats* the sender intends to pay
- `nostr` is the `9734 zap request` event, JSON encoded then URI encoded
- `lnurl` is the lnurl pay url of the recipient, encoded using bech32 with the prefix `lnurl`

This request should return a JSON response with a `pr` key, which is the invoice the sender must pay to finalize their zap. Here is an example flow in javascript:

```
const senderPubkey // The sender's pubkey
const recipientPubkey = // The recipient's pubkey
const callback = // The callback received from the recipients lnurl pay endpoint
const lnurl = // The recipient's lightning address, encoded as a lnurl
const sats = 21

const amount = sats * 1000
const relays = ['wss://nostr-pub.wellorder.net']
const event = encodeURI(JSON.stringify(await signEvent({
  kind: 9734,
  content: "",
  pubkey: senderPubkey,
  created_at: Math.round(Date.now() / 1000),
  tags: [
    ["relays", ...relays],
    ["amount", amount.toString()],
    ["lnurl", lnurl],
    ["p", recipientPubkey],
  ],
})))

const {pr: invoice} = await fetchJson(`${callback}?
amount=${amount}&nostr=${event}&lnurl=${lnurl}`)
```

**Appendix C: LNURL Server Configuration**

The lnurl server will need some additional pieces of information so that clients can know that zap invoices are supported:

1. Add a `nostrPubkey` to the lnurl-pay static endpoint `/.well-known/lnurlp/<user>`, where `nostrPubkey` is the nostr pubkey your server will use to sign `zap receipt` events. Clients will use this to validate `zap receipt`s.
2. Add an `allowsNostr` field and set it to true.

**Appendix D: LNURL Server Zap Request Validation**

When a client sends a `zap request` event to a server's lnurl-pay callback URL, there will be a `nostr` query parameter whose value is that event which is URI- and JSON-encoded. If present, the `zap request` event must be validated in the following ways:

1. It MUST have a valid nostr signature
2. It MUST have tags
3. It MUST have only one `p` tag
4. It MUST have 0 or 1 `e` tags
5. There should be a `relays` tag with the relays to send the `zap receipt` to.
6. If there is an `amount` tag, it MUST be equal to the `amount` query parameter.
7. If there is an `a` tag, it MUST be a valid event coordinate
8. There MUST be 0 or 1 `P` tags. If there is one, it MUST be equal to the `zap receipt`'s `pubkey`.

The event MUST then be stored for use later, when the invoice is paid.

**Appendix E: Zap Receipt Event**

A `zap receipt` is created by a lightning node when an invoice generated by a `zap request` is paid. `Zap receipt`s are only created when the invoice description (committed to the description hash) contains a `zap request` note.

When receiving a payment, the following steps are executed:

1. Get the description for the invoice. This needs to be saved somewhere during the generation of the description hash invoice. It is saved automatically for you with CLN, which is the reference implementation used here.
2. Parse the bolt11 description as a JSON nostr event. This SHOULD be validated based on the requirements in Appendix D, either when it is received, or before the invoice is paid.
3. Create a nostr event of kind `9735` as described below, and publish it to the `relays` declared in the `zap request`.

The following should be true of the `zap receipt` event:

- The `content` SHOULD be empty.
- The `created_at` date SHOULD be set to the invoice `paid_at` date for idempotency.
- `tags` MUST include the `p` tag (zap recipient) AND optional `e` tag from the `zap request` AND optional `a` tag from the `zap request` AND optional `P` tag from the pubkey of the zap request (zap sender).
- The `zap receipt` MUST have a `bolt11` tag containing the description hash bolt11 invoice.
- The `zap receipt` MUST contain a `description` tag which is the JSON-encoded zap request.
- `SHA256(description)` MUST match the description hash in the bolt11 invoice.
- The `zap receipt` MAY contain a `preimage` tag to match against the payment hash of the bolt11 invoice. This isn't really a payment proof, there is no real way to prove that the invoice is real or has been paid. You are trusting the author of the `zap receipt` for the legitimacy of the payment.

The `zap receipt` is not a proof of payment, all it proves is that some nostr user fetched an invoice. The existence of the `zap receipt` implies the invoice as paid, but it could be a lie given a rogue implementation.

A reference implementation for a zap-enabled lnurl server can be found here .

Example `zap receipt`:

```
{
    "id": "67b48a14fb66c60c8f9070bdeb37afdfcc3d08ad01989460448e4081eddda446",
    "pubkey": "9630f464cca6a5147aa8a35f0bcdd3ce485324e732fd39e09233b1d848238f31",
    "created_at": 1674164545,
    "kind": 9735,
    "tags": [
      ["p", "32e1827635450ebb3c5a7d12c1f8e7b2b514439ac10a67eef3d9fd9c5c68e245"],
      ["P", "97c70a44366a6535c145b333f973ea86dfdc2d7a99da618c40c64705ad98e322"],
      ["e", "3624762a1274dd9636e0c552b53086d70bc88c165bc4dc0f9e836a1eaf86c3b8"],
      ["bolt11",
"lnbc10u1p3unwfusp5t9r3yymhpfqculx78u027lxspgxcr2n2987mx2j55nnfs95nxnzqpp5jmrh92pfld78spqs78v9euf2385t

      ["description", "
{\"pubkey\":\"97c70a44366a6535c145b333f973ea86dfdc2d7a99da618c40c64705ad98e322\",\"content\":\"\",\"id
[[\"e\",\"3624762a1274dd9636e0c552b53086d70bc88c165bc4dc0f9e836a1eaf86c3b8\"],
[\"p\",\"32e1827635450ebb3c5a7d12c1f8e7b2b514439ac10a67eef3d9fd9c5c68e245\"],
[\"relays\",\"wss://relay.damus.io\",\"wss://nostr-
relay.wlvs.space\",\"wss://nostr.fmt.wiz.biz\",\"wss://relay.nostr.bg\",\"wss://nostr.oxtr.dev\",\"wss

      ["preimage",
"5d006d2cf1e73c7148e7519a4c68adc81642ce0e25a432b2434c99f97344c15f"]
    ],
    "content": "",
  }
```

**Appendix F: Validating Zap Receipts**

A client can retrieve `zap receipt`s on events and pubkeys using a NIP-01 filter, for example `{"kinds": [9735], "#e": [...]}`. Zaps MUST be validated using the following steps:

- The `zap receipt` event's `pubkey` MUST be the same as the recipient's lnurl provider's `nostrPubkey` (retrieved in step 1 of the protocol flow).
- The `invoiceAmount` contained in the `bolt11` tag of the `zap receipt` MUST equal the `amount` tag of the `zap request` (if present).
- The `lnurl` tag of the `zap request` (if present) SHOULD equal the recipient's `lnurl`.

**Appendix G: `zap` tag on other events**

When an event includes one or more `zap` tags, clients wishing to zap it SHOULD calculate the lnurl pay request based on the tags value instead of the event author's profile field. The tag's second argument is the `hex` string of the receiver's pub key and the third argument is the relay to download the receiver's metadata (Kind-0). An optional fourth parameter specifies the weight (a generalization of a percentage) assigned to the respective receiver. Clients should parse all weights, calculate a sum, and then a percentage to each receiver. If weights are not present, CLIENTS should equally divide the zap amount to all receivers. If weights are only partially present, receivers without a weight should not be zapped (`weight = 0`).

```
{
    "tags": [
        [ "zap", "82341f882b6eabcd2ba7f1ef90aad961cf074af15b9ef44a09f9d2a8fbfbe6a2",
```

```
"wss://nostr.oxtr.dev", "1" ],  // 25%
        [ "zap", "fa984bd7dbb282f07e16e7ae87b26a2a7b9b90b7246a44771f0cf5ae58018f52",
"wss://nostr.wine/",    "1" ],  // 25%
        [ "zap", "460c25e682fda7832b52d1f22d3d22b3176d972f60dcdc3212ed8c92ef85065c",
"wss://nos.lol/",       "2" ]   // 50%
    ]
}
```

Clients MAY display the zap split configuration in the note.

### Future Work

Zaps can be extended to be more private by encrypting `zap request` notes to the target user, but for simplicity it has been left out of this initial draft.

## NIP-58

## Badges

`draft` `optional`

Three special events are used to define, award and display badges in user profiles:

1. A “Badge Definition” event is defined as an addressable event with kind `30009` having a `d` tag with a value that uniquely identifies the badge (e.g. `bravery`) published by the badge issuer. Badge definitions can be updated.
2. A “Badge Award” event is a kind `8` event with a single `a` tag referencing a “Badge Definition” event and one or more `p` tags, one for each pubkey the badge issuer wishes to award. Awarded badges are immutable and non-transferrable.
3. A “Profile Badges” event is defined as an *addressable event* with kind `30008` with a `d` tag with the value `profile_badges`. Profile badges contain an ordered list of pairs of `a` and `e` tags referencing a `Badge Definition` and a `Badge Award` for each badge to be displayed.

#### Badge Definition event

The following tags MUST be present:

- `d` tag with the unique name of the badge.

The following tags MAY be present:

- A `name` tag with a short name for the badge.
- `image` tag whose value is the URL of a high-resolution image representing the badge. The second value optionally specifies the dimensions of the image as `width`x`height` in pixels. Badge recommended dimensions is 1024x1024 pixels.
- A `description` tag whose value MAY contain a textual representation of the image, the meaning behind the badge, or the reason of its issuance.
- One or more `thumb` tags whose first value is an URL pointing to a thumbnail version of the image referenced in the `image` tag. The second value optionally specifies the dimensions of the thumbnail as `width`x`height` in pixels.

#### Badge Award event

The following tags MUST be present:

- An `a` tag referencing a kind `30009` Badge Definition event.
- One or more `p` tags referencing each pubkey awarded.

#### Profile Badges Event

The number of badges a pubkey can be awarded is unbounded. The Profile Badge event allows individual users to accept or reject awarded badges, as well as choose the display order of badges on their profiles.

The following tags MUST be present:

- A `d` tag with the unique identifier `profile_badges`

The following tags MAY be present:

- Zero or more ordered consecutive pairs of `a` and `e` tags referencing a kind `30009` Badge Definition and kind `8` Badge Award, respectively. Clients SHOULD ignore `a` without corresponding `e` tag and viceversa. Badge Awards referenced by the `e` tags should contain the same `a` tag.

**Motivation**

Users MAY be awarded badges (but not limited to) in recognition, in gratitude, for participation, or in appreciation of a certain goal, task or cause.

Users MAY choose to decorate their profiles with badges for fame, notoriety, recognition, support, etc., from badge issuers they deem reputable.

**Recommendations**

Clients MAY whitelist badge issuers (pubkeys) for the purpose of ensuring they retain a valuable/special factor for their users.

Badge image recommended aspect ratio is 1:1 with a high-res size of 1024x1024 pixels.

Badge thumbnail image recommended dimensions are: 512x512 (xl), 256x256 (l), 64x64 (m), 32x32 (s) and 16x16 (xs).

Clients MAY choose to render less badges than those specified by users in the Profile Badges event or replace the badge image and thumbnails with ones that fits the theme of the client.

Clients SHOULD attempt to render the most appropriate badge thumbnail according to the number of badges chosen by the user and space available. Clients SHOULD attempt render the high-res version on user action (click, tap, hover).

**Example of a Badge Definition event**

```
{
  "pubkey": "alice",
  "kind": 30009,
  "tags": [
    ["d", "bravery"],
    ["name", "Medal of Bravery"],
    ["description", "Awarded to users demonstrating bravery"],
    ["image", "https://nostr.academy/awards/bravery.png", "1024x1024"],
    ["thumb", "https://nostr.academy/awards/bravery_256x256.png", "256x256"]
  ],
  // other fields...
}
```

**Example of Badge Award event**

```
{
  "id": "<badge award event id>",
  "kind": 8,
  "pubkey": "alice",
  "tags": [
    ["a", "30009:alice:bravery"],
    ["p", "bob", "wss://relay"],
    ["p", "charlie", "wss://relay"]
  ],
  // other fields...
}
```

**Example of a Profile Badges event**

Honorable Bob The Brave:

```
{
  "kind": 30008,
  "pubkey": "bob",
  "tags": [
    ["d", "profile_badges"],
    ["a", "30009:alice:bravery"],
    ["e", "<bravery badge award event id>", "wss://nostr.academy"],
```

```
    ["a", "30009:alice:honor"],
    ["e", "<honor badge award event id>", "wss://nostr.academy"]
  ],
  // other fields...
}
```

## NIP-59

## Gift Wrap

`optional`

This NIP defines a protocol for encapsulating any nostr event. This makes it possible to obscure most metadata for a given event, perform collaborative signing, and more.

This NIP *does not* define any messaging protocol. Applications of this NIP should be defined separately.

This NIP relies on NIP-44 's versioned encryption algorithms.

## Overview

This protocol uses three main concepts to protect the transmission of a target event: `rumor`s, `seal`s, and `gift wrap`s.

- A `rumor` is a regular nostr event, but is **not signed**. This means that if it is leaked, it cannot be verified.
- A `rumor` is serialized to JSON, encrypted, and placed in the `content` field of a `seal`. The `seal` is then signed by the author of the note. The only information publicly available on a `seal` is who signed it, but not what was said.
- A `seal` is serialized to JSON, encrypted, and placed in the `content` field of a `gift wrap`.

This allows the isolation of concerns across layers:

- A rumor carries the content but is unsigned, which means if leaked it will be rejected by relays and clients, and can't be authenticated. This provides a measure of deniability.
- A seal identifies the author without revealing the content or the recipient.
- A gift wrap can add metadata (recipient, tags, a different author) without revealing the true author.

## Protocol Description

### 1. The Rumor Event Kind

A `rumor` is the same thing as an unsigned event. Any event kind can be made a `rumor` by removing the signature.

### 2. The Seal Event Kind

A `seal` is a `kind:13` event that wraps a `rumor` with the sender's regular key. The `seal` is **always** encrypted to a receiver's pubkey but there is no `p` tag pointing to the receiver. There is no way to know who the rumor is for without the receiver's or the sender's private key. The only public information in this event is who is signing it.

```
{
  "id": "<id>",
  "pubkey": "<real author's pubkey>",
  "content": "<encrypted rumor>",
  "kind": 13,
  "created_at": 1686840217,
  "tags": [],
  "sig": "<real author's pubkey signature>"
}
```

Tags MUST must always be empty in a `kind:13`. The inner event MUST always be unsigned.

### 3. Gift Wrap Event Kind

A `gift wrap` event is a `kind:1059` event that wraps any other event. `tags` SHOULD include any information needed to route the event to its intended recipient, including the recipient's `p` tag or NIP-13 proof of work.

```
{
  "id": "<id>",
  "pubkey": "<random, one-time-use pubkey>",
  "content": "<encrypted kind 13>",
```

```
  "kind": 1059,
  "created_at": 1686840217,
  "tags": [["p", "<recipient pubkey>"]],
  "sig": "<random, one-time-use pubkey signature>"
}
```

## Encrypting Payloads

Encryption is done following NIP-44 on the JSON-encoded event. Place the encryption payload in the `.content` of the wrapper event (either a `seal` or a `gift wrap`).

## Other Considerations

If a `rumor` is intended for more than one party, or if the author wants to retain an encrypted copy, a single `rumor` may be wrapped and addressed for each recipient individually.

The canonical `created_at` time belongs to the `rumor`. All other timestamps SHOULD be tweaked to thwart time-analysis attacks. Note that some relays don't serve events dated in the future, so all timestamps SHOULD be in the past.

Relays may choose not to store gift wrapped events due to them not being publicly useful. Clients MAY choose to attach a certain amount of proof-of-work to the wrapper event per NIP-13 in a bid to demonstrate that the event is not spam or a denial-of-service attack.

To protect recipient metadata, relays SHOULD guard access to `kind 1059` events based on user AUTH. When possible, clients should only send wrapped events to relays that offer this protection.

To protect recipient metadata, relays SHOULD only serve `kind 1059` events intended for the marked recipient. When possible, clients should only send wrapped events to `read` relays for the recipient that implement AUTH, and refuse to serve wrapped events to non-recipients.

## An Example

Let's send a wrapped `kind 1` message between two parties asking "Are you going to the party tonight?"

- Author private key: `0beebd062ec8735f4243466049d7747ef5d6594ee838de147f8aab842b15e273`
- Recipient private key: `e108399bd8424357a710b606ae0c13166d853d327e47a6e5e038197346bdbf45`
- Ephemeral wrapper key: `4f02eac59266002db5801adc5270700ca69d5b8f761d8732fab2fbf233c90cbd`

Note that this messaging protocol should not be used in practice, this is just an example. Refer to other NIPs for concrete messaging protocols that depend on gift wraps.

### 1. Create an event

Create a `kind 1` event with the message, the receivers, and any other tags you want, signed by the author. Do not sign the event.

```
{
  "created_at": 1691518405,
  "content": "Are you going to the party tonight?",
  "tags": [],
  "kind": 1,
  "pubkey": "611df01bfcf85c26ae65453b772d8f1dfd25c264621c0277e1fc1518686faef9",
  "id": "9dd003c6d3b73b74a85a9ab099469ce251653a7af76f523671ab828acd2a0ef9"
}
```

### 2. Seal the rumor

Encrypt the JSON-encoded `rumor` with a conversation key derived using the author's private key and the recipient's public key. Place the result in the `content` field of a `kind 13 seal` event. Sign it with the author's key.

```
{
  "content":
"AqBCdwoS7/tPK+QGkPCadJTn8FxGkd24iAp03BR9/M0uw6n4RFAFSPAKKMgkzVMoRyR3ZS/aqATDFvoZJOkE9cPG/TAzmyZvr/WUI

  "kind": 13,
  "created_at": 1703015180,
  "pubkey": "611df01bfcf85c26ae65453b772d8f1dfd25c264621c0277e1fc1518686faef9",
```

```
  "tags": [],
  "id": "28a87d7c074d94a58e9e89bb3e9e4e813e2189f285d797b1c56069d36f59eaa7",
  "sig":
"02fc3facf6621196c32912b1ef53bac8f8bfe9db51c0e7102c073103586b0d29c3f39bdaa1e62856c20e90b6c7cc5dc34ca8b

}
```

### 3. Wrap the seal

Encrypt the JSON-encoded `kind 13` event with your ephemeral, single-use random key. Place the result in the `content` field of a `kind 1059`. Add a single `p` tag containing the recipient's public key. Sign the `gift wrap` using the random key generated in the previous step.

```
{
  "content":
"AhC3Qj/QsKJFWuf6xroiYip+2yK95qPwJjVvFujhzSguJWb/6TlPpBW0CGFwfufCs2Zyb0JeuLmZhNlnqecAAalC4ZCugB+I9ViA5

  "kind": 1059,
  "created_at": 1703021488,
  "pubkey": "18b1a75918f1f2c90c23da616bce317d36e348bcf5f7ba55e75949319210c87c",
  "id": "5c005f3ccf01950aa8d131203248544fb1e41a0d698e846bd419cec3890903ac",
  "sig":
"35fabdae4634eb630880a1896a886e40fd6ea8a60958e30b89b33a93e6235df750097b04f9e13053764251b8bc5dd7e8e0794

  "tags": [["p",
"166bf3765ebd1fc55decfe395beff2ea3b2a4e0a8946e7eb578512b555737c99"]],
}
```

### 4. Broadcast Selectively

Broadcast the `kind 1059` event to the recipient's relays only. Delete all the other events.

## Code Samples

### JavaScript

```
import {bytesToHex} from "@noble/hashes/utils"
import type {EventTemplate, UnsignedEvent, Event} from "nostr-tools"
import {getPublicKey, getEventHash, nip19, nip44, finalizeEvent, generateSecretKey}
from "nostr-tools"

type Rumor = UnsignedEvent & {id: string}

const TWO_DAYS = 2 * 24 * 60 * 60

const now = () => Math.round(Date.now() / 1000)
const randomNow = () => Math.round(now() - (Math.random() * TWO_DAYS))

const nip44ConversationKey = (privateKey: Uint8Array, publicKey: string) =>
  nip44.v2.utils.getConversationKey(bytesToHex(privateKey), publicKey)

const nip44Encrypt = (data: EventTemplate, privateKey: Uint8Array, publicKey:
string) =>
  nip44.v2.encrypt(JSON.stringify(data), nip44ConversationKey(privateKey,
publicKey))

const nip44Decrypt = (data: Event, privateKey: Uint8Array) =>
  JSON.parse(nip44.v2.decrypt(data.content, nip44ConversationKey(privateKey,
data.pubkey)))

const createRumor = (event: Partial<UnsignedEvent>, privateKey: Uint8Array) => {
  const rumor = {
    created_at: now(),
    content: "",
    tags: [],
    ...event,
    pubkey: getPublicKey(privateKey),
  } as any
```

```
  rumor.id = getEventHash(rumor)

  return rumor as Rumor
}

const createSeal = (rumor: Rumor, privateKey: Uint8Array, recipientPublicKey:
string) => {
  return finalizeEvent(
    {
      kind: 13,
      content: nip44Encrypt(rumor, privateKey, recipientPublicKey),
      created_at: randomNow(),
      tags: [],
    },
    privateKey
  ) as Event
}

const createWrap = (event: Event, recipientPublicKey: string) => {
  const randomKey = generateSecretKey()

  return finalizeEvent(
    {
      kind: 1059,
      content: nip44Encrypt(event, randomKey, recipientPublicKey),
      created_at: randomNow(),
      tags: [["p", recipientPublicKey]],
    },
    randomKey
  ) as Event
}

// Test case using the above example
const senderPrivateKey =
nip19.decode(`nsec1p0ht6p3wepe47sjrgesyn4m50m6avk2waqudu9rl324cg2c4ufesyp6rdg`).data
const recipientPrivateKey =
nip19.decode(`nsec1uyyrnx7cgfp40fcskcr2urqnzekc20fj0er6de0q8qvhx34ahazsvs9p36`).data
const recipientPublicKey = getPublicKey(recipientPrivateKey)

const rumor = createRumor(
  {
    kind: 1,
    content: "Are you going to the party tonight?",
  },
  senderPrivateKey
)

const seal = createSeal(rumor, senderPrivateKey, recipientPublicKey)
const wrap = createWrap(seal, recipientPublicKey)

// Recipient unwraps with their private key.

const unwrappedSeal = nip44Decrypt(wrap, recipientPrivateKey)
const unsealedRumor = nip44Decrypt(unwrappedSeal, recipientPrivateKey)
```

## NIP-60

### Cashu Wallet

`draft` `optional`

This NIP defines the operations of a cashu-based wallet.

A cashu wallet is a wallet which information is stored in relays to make it accessible across applications.

The purpose of this NIP is:

- ease-of-use: new users immediately are able to receive funds without creating accounts with other services.
- interoperability: users' wallets follows them across applications.

This NIP doesn't deal with users' *receiving* money from someone else, it's just to keep state of the user's wallet.

## High-level flow

1. A user has a `kind:37375` event that represents a wallet.
2. A user has `kind:7375` events that represent the unspent proofs of the wallet. – The proofs are encrypted with the user's private key.
3. A user has `kind:7376` events that represent the spending history of the wallet – This history is for informational purposes only and is completely optional.

### Wallet Event

```
{
    "kind": 37375,
    "content": nip44_encrypt([
        [ "balance", "100", "sat" ],
        [ "privkey", "hexkey" ] // explained in NIP-61
    ]),
    "tags": [
        [ "d", "my-wallet" ],
        [ "mint", "https://mint1" ],
        [ "mint", "https://mint2" ],
        [ "mint", "https://mint3" ],
        [ "name", "my shitposting wallet" ],
        [ "unit", "sat" ],
        [ "description", "a wallet for my day-to-day shitposting" ],
        [ "relay", "wss://relay1" ],
        [ "relay", "wss://relay2" ],
    ]
}
```

The wallet event is a parameterized replaceable event `kind:37375`.

Tags:

- `d` - wallet ID.
- `mint` - Mint(s) this wallet uses – there MUST be one or more mint tags.
- `relay` - Relays where the wallet and related events can be found. – one ore more relays SHOULD be specified. If missing, clients should follow [[NIP-65]].
- `unit` - Base unit of the wallet (e.g. "sat", "usd", etc).
- `name` - Optional human-readable name for the wallet.
- `description` - Optional human-readable description of the wallet.
- `balance` - Optional best-effort balance of the wallet that can serve as a placeholder while an accurate balance is computed from fetching all unspent proofs.
- `privkey` - Private key used to unlock P2PK ecash. MUST be stored encrypted in the `.content` field. **This is a different private key exclusively used for the wallet, not associated in any way to the user's nostr private key** – This is only used when receiving funds from others, described in NIP-61.

Any tag, other than the `d` tag, can be [[NIP-44]] encrypted into the `.content` field.

#### Deleting a wallet event

Due to PRE being hard to delete, if a user wants to delete a wallet, they should empty the event and keep just the `d` identifier and add a `deleted` tag.

### Token Event

Token events are used to record the unspent proofs that come from the mint.

There can be multiple `kind:7375` events for the same mint, and multiple proofs inside each `kind:7375` event.

```
{
    "kind": 7375,
    "content": nip44_encrypt({
        "mint": "https://stablenut.umint.cash",
        "proofs": [
```

```
            {
                "id": "005c2502034d4f12",
                "amount": 1,
                "secret": "z+zyxAVLRqN9lEjxuNPSyRJzEstbl69Jc1vtimvtkPg=",
                "C":
"0241d98a8197ef238a192d47edf191a9de78b657308937b4f7dd0aa53beae72c46"
            }
        ]
    }),
    "tags": [
        [ "a", "37375:<pubkey>:my-wallet" ]
    ]
}
```

`.content` is a [[NIP-44]] encrypted payload storing the mint and the unencoded proofs.

- `a` an optional tag linking the token to a specific wallet.

#### Spending proofs

When one or more proofs of a token are spent, the token event should be [[NIP-09]]-deleted and, if some proofs are unspent from the same token event, a new token event should be created rolling over the unspent proofs and adding any change outputs to the new token event.

### Spending History Event

Clients SHOULD publish `kind:7376` events to create a transaction history when their balance changes.

```
{
    "kind": 7376,
    "content": nip44_encrypt([
        [ "direction", "in" ], // in = received, out = sent
        [ "amount", "1", "sat" ],
        [ "e", "<event-id-of-spent-token>", "<relay-hint>", "created" ],
    ]),
    "tags": [
        [ "a", "37375:<pubkey>:my-wallet" ],
    ]
}
```

- `direction` - The direction of the transaction; `in` for received funds, `out` for sent funds.
- `a` - The wallet the transaction is related to.

Clients MUST add `e` tags to create references of destroyed and created token events along with the marker of the meaning of the tag:

- `created` - A new token event was created.
- `destroyed` - A token event was destroyed.
- `redeemed` - A [[NIP-61]] nutzap was redeemed.

All tags can be [[NIP-44]] encrypted. Clients SHOULD leave `e` tags with a `redeemed` marker unencrypted.

Multiple `e` tags can be added to a `kind:7376` event.

## Flow

A client that wants to check for user's wallets information starts by fetching `kind:10019` events from the user's relays, if no event is found, it should fall back to using the user's [[NIP-65]] relays.

### Fetch wallet and token list

From those relays, the client should fetch wallet and token events.

`"kinds": [37375, 7375], "authors": ["<my-pubkey>"]`

### Fetch proofs

While the client is fetching (and perhaps validating) proofs it can use the optional `balance` tag of the wallet event to display a estimate of the balance of the wallet.

### Spending token

If Alice spends 4 sats from this token event

```
{
    "kind": 7375,
    "id": "event-id-1",
    "content": nip44_encrypt({
        "mint": "https://stablenut.umint.cash",
        "proofs": [
            { "id": "1", "amount": 1 },
            { "id": "2", "amount": 2 },
            { "id": "3", "amount": 4 },
            { "id": "4", "amount": 8 },
        ]
    }),
    "tags": [
        [ "a", "37375:<pubkey>:my-wallet" ]
    ]
}
```

Her client:

- MUST roll over the unspent proofs:

```
{
    "kind": 7375,
    "id": "event-id-2",
    "content": nip44_encrypt({
        "mint": "https://stablenut.umint.cash",
        "proofs": [
            { "id": "1", "amount": 1 },
            { "id": "2", "amount": 2 },
            { "id": "4", "amount": 8 },
        ]
    }),
    "tags": [
        [ "a", "37375:<pubkey>:my-wallet" ]
    ]
}
```

- MUST delete event `event-id-1`
- SHOULD create a `kind:7376` event to record the spend

```
{
    "kind": 7376,
    "content": nip44_encrypt([
        [ "direction", "out" ],
        [ "amount", "4", "sats" ],
        [ "e", "<event-id-1>", "<relay-hint>", "destroyed" ],
        [ "e", "<event-id-2>", "<relay-hint>", "created" ],
    ]),
    "tags": [
        [ "a", "37375:<pubkey>:my-wallet" ],
    ]
}
```

### Redeeming a quote (optional)

When creating a quote at a mint, an event can be used to keep the state of the quote ID, which will be used to check when the quote has been paid. These events should be created with an expiration tag [[NIP-40]] matching the expiration of the bolt11 received from the mint; this signals to relays when they can safely discard these events.

Application developers are encouraged to use local state when possible and only publish this event when it makes sense in the context of their application.

```
{
    "kind": 7374,
    "content": nip44_encrypt("quote-id"),
```

```
    "tags": [
        [ "expiration", "<expiration-timestamp>" ],
        [ "mint", "<mint-url>" ],
        [ "a", "37375:<pubkey>:my-wallet" ]
    ]
}
```

### Appendix 1: Validating proofs

Clients can optionally validate proofs to make sure they are not working from an old state; this logic is left up to particular implementations to decide when and why to do it, but if some proofs are checked and deemed to have been spent, the client should delete the token and roll over any unspent proof.

## NIP-61:

### Nut Zaps

A Nut Zap is a P2PK cashu token where the payment itself is the receipt.

## High-level flow

Alice wants to nutzap 1 sat to Bob because of an event `event-id-1` she liked.

### Alice nutzaps Bob

1. Alice fetches event `kind:10019` from Bob to see the mints Bob trusts.
2. She mints a token at that mint (or swaps some tokens she already had in that mint) p2pk-locked to the pubkey Bob has listed in his `kind:10019`.
3. She publishes a `kind:9321` event to the relays Bob indicated with the proofs she minted.

### Bob receives the nutzap

1. At some point, Bob's client fetches `kind:9321` events p-tagging him from his relays.
2. Bob's client swaps the token into his wallet.

## Nutzap informational event

```
{
    "kind": 10019,
    "tags": [
        [ "relay", "wss://relay1" ],
        [ "relay", "wss://relay2" ],
        [ "mint", "https://mint1", "usd", "sat" ],
        [ "mint", "https://mint2", "sat" ],
        [ "pubkey", "<p2pk-pubkey>" ]
    ]
}
```

`kind:10019` is an event that is useful for others to know how to send money to the user.

- `relay` - Relays where the user will be reading token events from. If a user wants to send money to the user, they should write to these relays.
- `mint` - Mints the user is explicitly agreeing to use to receive funds on. Clients SHOULD not send money on mints not listed here or risk burning their money. Additional markers can be used to list the supported base units of the mint.
- `pubkey` - Pubkey that SHOULD be used to P2PK-lock receiving nutzaps. If not present, clients SHOULD use the pubkey of the recipient. This is explained in Appendix 1.

### Nutzap event

Event `kind:9321` is a nutzap event published by the sender, p-tagging the recipient. The outputs are P2PK-locked to the pubkey the recipient indicated in their `kind:10019` event or to the recipient pubkey if the `kind:10019` event doesn't have a explicit pubkey.

Clients MUST prefix the pubkey they p2pk-lock with `"02"` (for nostr<>cashu pubkey compatibility).

```
{
    kind: 9321,
    content: "Thanks for this great idea.",
```

```
    pubkey: "sender-pubkey",
    tags: [
        [ "amount", "1" ],
        [ "unit", "sat" ],
        [ "proof", "
{\"amount\":1,\"C\":\"02277c66191736eb72fce9d975d08e3191f8f96afb73ab1eec37e4465683066d3f\",\"id\":\"00
[\\\"P2PK\\\",
{\\\"nonce\\\":\\\"b00bdd0467b0090a25bdf2d2f0d45ac4e355c482c1418350f273a04fedaaee83\\\",\\\"data\\\":\'
 ],
        [ "u", "https://stablenut.umint.cash", ],
        [ "e", "<zapped-event-id>", "<relay-hint>" ],
        [ "p", "e9fbced3a42dcf551486650cc752ab354347dd413b307484e4fd1818ab53f991" ],
// recipient of nut zap
    ]
}
```

- `.content` is an optional comment for the nutzap
- `amount` is a shorthand for the combined amount of all outputs. – Clients SHOULD validate that the sum of the amounts in the outputs matches.
- `unit` is the base unit of the amount.
- `proof` is one ore more proofs p2pk-locked to the pubkey the recipient specified in their `kind:10019` event.
- `u` is the mint the URL of the mint EXACTLY as specified by the recipient's `kind:10019`.
- `e` zero or one event that is being nutzapped.
- `p` exactly one pubkey, specifying the recipient of the nutzap.

WIP: Clients SHOULD embed a DLEQ proof in the nutzap event to make it possible to verify nutzaps without talking to the mint.

## Sending a nutzap

- The sender fetches the recipient's `kind:10019`.
- The sender mints/swaps ecash on one of the recipient's listed mints.
- The sender p2pk locks to the recipient's specified pubkey in their

## Receiving nutzaps

Clients should REQ for nut zaps:

- Filtering with `#u` for mints they expect to receive ecash from.
    - this is to prevent even interacting with mints the user hasn't explicitly signaled.
- Filtering with `since` of the most recent `kind:7376` event the same user has created.
    - this can be used as a marker of the nut zaps that have already been swaped by the user – clients might choose to use other kinds of markers, including internal state – this is just a guidance of one possible approach.

Clients MIGHT choose to use some kind of filtering (e.g. WoT) to ignore spam.

`{ "kinds": [9321], "#p": "my-pubkey", "#u": [ "<mint-1>", "<mint-2>"], "since": <latest-created_at-of-kind-7376> }`.

Upon receiving a new nut zap, the client should swap the tokens into a wallet the user controls, either a [[NIP-60]] wallet, their own LN wallet or anything else.

### Updating nutzap-redemption history

When claiming a token the client SHOULD create a `kind:7376` event and `e` tag the original nut zap event. This is to record that this token has already been claimed (and shouldn't be attempted again) and as signaling to the recipient that the ecash has been redeemed.

Multiple `kind:9321` events can be tagged in the same `kind:7376` event.

```
{
    "kind": 7376,
    "content": nip44_encrypt([
        [ "direction", "in" ], // in = received, out = sent
        [ "amount", "1", "sat" ],
        [ "e", "<7375-event-id>", "relay-hint", "created" ] // new token event that
was created
    ]),
```

```
    "tags": [
        [ "a", "37375:<pubkey>:my-wallet" ], // an optional wallet tag
        [ "e", "<9321-event-id>", "relay-hint", "redeemed" ], // nutzap event that
has been redeemed
        [ "p", "sender-pubkey" ] // pubkey of the author of the 9321 event (nutzap
sender)
    ]
}
```

Events that redeem a nutzap SHOULD be published to the sender's [[NIP-65]] relays.

### Verifying a Cashu Zap

- Clients SHOULD check that the receiving user has issued a `kind:10019` tagging the mint where the cashu has been minted.
- Clients SHOULD check that the token is locked to the pubkey the user has listed in their `kind:10019`.

### Final Considerations

1. Clients SHOULD guide their users to use NUT-11 (P2PK) compatible-mints in their `kind:10019` event to avoid receiving nut zaps anyone can spend
2. Clients SHOULD normalize and deduplicate mint URLs as described in NIP-65.
3. A nut zap MUST be sent to a mint the recipient has listed in their `kind:10019` event or to the NIP-65 relays of the recipient, failure to do so may result in the recipient donating the tokens to the mint since the recipient might never see the event.

### Appendix 1: Alternative P2PK pubkey

Clients might not have access to the user's private key (i.e. NIP-07, NIP-46 signing) and, as such, the private key to sign cashu spends might not be available, which would make spending the P2PK incoming nutzaps impossible.

For this scenarios clients can:

- add a `pubkey` tag to the `kind:10019` (indicating which pubkey senders should P2PK to)
- store the private key in the `kind:37375` event in the nip44-encrypted `content` field.

## This is to avoid depending on NIP-07/46 adaptations to sign cashu payloads. NIP-64

## Chess (Portable Game Notation)

`draft` `optional`

This NIP defines `kind:64` notes representing chess games in PGN format, which can be read by humans and is also supported by most chess software.

### Note

#### Content

The `.content` of these notes is a string representing a PGN-database .

#### Notes

```
{
  "kind": 64,
  "content": "1. e4 *",
  // other fields...
}
```

```
{
  "kind": 64,
  "tags": [
    ["alt", "Fischer vs. Spassky in Belgrade on 1992-11-04 (F/S Return Match, Round
29)"],
    // rest of tags...
  ],
  "content": "[Event \"F/S Return Match\"]\n[Site \"Belgrade, Serbia JUG\"]\n[Date
```

```
\"1992.11.04\"]\n[Round \"29\"]\n[White \"Fischer, Robert J.\"]\n[Black \"Spassky,
Boris V.\"]\n[Result \"1/2-1/2\"]\n\n1. e4 e5 2. Nf3 Nc6 3. Bb5 {This opening is
called the Ruy Lopez.} 3... a6\n4. Ba4 Nf6 5. O-O Be7 6. Re1 b5 7. Bb3 d6 8. c3 O-O
9. h3 Nb8 10. d4 Nbd7\n11. c4 c6 12. cxb5 axb5 13. Nc3 Bb7 14. Bg5 b4 15. Nb1 h6 16.
Bh4 c5 17. dxe5\nNxe4 18. Bxe7 Qxe7 19. exd6 Qf6 20. Nbd2 Nxd6 21. Nc4 Nxc4 22. Bxc4
Nb6\n23. Ne5 Rae8 24. Bxf7+ Rxf7 25. Nxf7 Rxe1+ 26. Qxe1 Kxf7 27. Qe3 Qg5 28.
Qxg5\nhxg5 29. b3 Ke6 30. a3 Kd6 31. axb4 cxb4 32. Ra5 Nd5 33. f3 Bc8 34. Kf2
Bf5\n35. Ra7 g6 36. Ra6+ Kc5 37. Ke1 Nf4 38. g3 Nxh3 39. Kd2 Kb5 40. Rd6 Kc5 41.
Ra6\nNf2 42. g4 Bd3 43. Re6 1/2-1/2",
  // other fields...
}
```

### Client Behavior

Clients SHOULD display the content represented as chessboard.

Clients SHOULD publish PGN notes in "export format" ("strict mode", i.e. created by machines) but expect incoming notes to be in "import format" ("lax mode", i.e. created by humans).

Clients SHOULD check whether the formatting is valid and all moves comply with chess rules.

Clients MAY include additional tags (e.g. like `"alt"` ) in order to represent the note to users of non-supporting clients.

### Relay Behavior

Relays MAY validate PGN contents and reject invalid notes.

### Examples

```
// A game where nothing is known. Game still in progress, game abandoned, or result
otherwise unknown.
// Maybe players died before a move has been made.
*
```

```
1. e4 *
```

```
[White "Fischer, Robert J."]
[Black "Spassky, Boris V."]

1. e4 e5 2. Nf3 Nc6 3. Bb5 {This opening is called the Ruy Lopez.} *
```

```
[Event "F/S Return Match"]
[Site "Belgrade, Serbia JUG"]
[Date "1992.11.04"]
[Round "29"]
[White "Fischer, Robert J."]
[Black "Spassky, Boris V."]
[Result "1/2-1/2"]

1. e4 e5 2. Nf3 Nc6 3. Bb5 {This opening is called the Ruy Lopez.} 3... a6
4. Ba4 Nf6 5. O-O Be7 6. Re1 b5 7. Bb3 d6 8. c3 O-O 9. h3 Nb8 10. d4 Nbd7
11. c4 c6 12. cxb5 axb5 13. Nc3 Bb7 14. Bg5 b4 15. Nb1 h6 16. Bh4 c5 17. dxe5
Nxe4 18. Bxe7 Qxe7 19. exd6 Qf6 20. Nbd2 Nxd6 21. Nc4 Nxc4 22. Bxc4 Nb6
23. Ne5 Rae8 24. Bxf7+ Rxf7 25. Nxf7 Rxe1+ 26. Qxe1 Kxf7 27. Qe3 Qg5 28. Qxg5
hxg5 29. b3 Ke6 30. a3 Kd6 31. axb4 cxb4 32. Ra5 Nd5 33. f3 Bc8 34. Kf2 Bf5
35. Ra7 g6 36. Ra6+ Kc5 37. Ke1 Nf4 38. g3 Nxh3 39. Kd2 Kb5 40. Rd6 Kc5 41. Ra6
Nf2 42. g4 Bd3 43. Re6 1/2-1/2
```

```
[Event "Hourly HyperBullet Arena"]
[Site "https://lichess.org/wxx4GldJ"]
[Date "2017.04.01"]
[White "T_LUKE"]
[Black "decidement"]
[Result "1-0"]
[UTCDate "2017.04.01"]
[UTCTime "11:56:14"]
[WhiteElo "2047"]
```

```
[BlackElo "1984"]
[WhiteRatingDiff "+10"]
[BlackRatingDiff "-7"]
[Variant "Standard"]
[TimeControl "30+0"]
[ECO "B00"]
[Termination "Abandoned"]

1. e4 1-0


[Event "Hourly HyperBullet Arena"]
[Site "https://lichess.org/rospUdSk"]
[Date "2017.04.01"]
[White "Bastel"]
[Black "oslochess"]
[Result "1-0"]
[UTCDate "2017.04.01"]
[UTCTime "11:55:56"]
[WhiteElo "2212"]
[BlackElo "2000"]
[WhiteRatingDiff "+6"]
[BlackRatingDiff "-4"]
[Variant "Standard"]
[TimeControl "30+0"]
[ECO "A01"]
[Termination "Normal"]

1. b3 d5 2. Bb2 c6 3. Nc3 Bf5 4. d4 Nf6 5. e3 Nbd7 6. f4 Bg6 7. Nf3 Bh5 8. Bd3 e6 9.
O-O Be7 10. Qe1 O-O 11. Ne5 Bg6 12. Nxg6 hxg6 13. e4 dxe4 14. Nxe4 Nxe4 15. Bxe4 Nf6
16. c4 Bd6 17. Bc2 Qc7 18. f5 Be7 19. fxe6 fxe6 20. Qxe6+ Kh8 21. Qh3+ Kg8 22. Bxg6
Qd7 23. Qe3 Bd6 24. Bf5 Qe7 25. Be6+ Kh8 26. Qh3+ Nh7 27. Bf5 Rf6 28. Qxh7# 1-0
```

**Resources**

## NIP-65

## Relay List Metadata

`draft` `optional`

Defines a replaceable event using `kind:10002` to advertise preferred relays for discovering a user's content and receiving fresh content from others.

The event MUST include a list of `r` tags with relay URIs and a `read` or `write` marker. Relays marked as `read` / `write` are called READ / WRITE relays, respectively. If the marker is omitted, the relay is used for both purposes.

The `.content` is not used.

```
{
  "kind": 10002,
  "tags": [
    ["r", "wss://alicerelay.example.com"],
    ["r", "wss://brando-relay.com"],
    ["r", "wss://expensive-relay.example2.com", "write"],
    ["r", "wss://nostr-relay.example.com", "read"]
  ],
  "content": "",
  // other fields...
}
```

This NIP doesn't fully replace relay lists that are designed to configure a client's usage of relays (such as `kind:3` style relay lists). Clients MAY use other relay lists in situations where a `kind:10002` relay list cannot be found.

### When to Use Read and Write Relays

When seeking events **from** a user, Clients SHOULD use the WRITE relays of the user's `kind:10002`.

When seeking events **about** a user, where the user was tagged, Clients SHOULD use the READ relays of the user's `kind:10002`.

When broadcasting an event, Clients SHOULD:

- Broadcast the event to the WRITE relays of the author
- Broadcast the event to all READ relays of each tagged user

### Motivation

The old model of using a fixed relay list per user centralizes in large relay operators:

- Most users submit their posts to the same highly popular relays, aiming to achieve greater visibility among a broader audience
- Many users are pulling events from a large number of relays in order to get more data at the expense of duplication
- Events are being copied between relays, oftentimes to many different relays

This NIP allows Clients to connect directly with the most up-to-date relay set from each individual user, eliminating the need of broadcasting events to popular relays.

### Final Considerations

1. Clients SHOULD guide users to keep `kind:10002` lists small (2-4 relays).
2. Clients SHOULD spread an author's `kind:10002` event to as many relays as viable.
3. `kind:10002` events should primarily be used to advertise the user's preferred relays to others. A user's own client may use other heuristics for selecting relays for fetching data.
4. DMs SHOULD only be broadcasted to the author's WRITE relays and to the receiver's READ relays to keep maximum privacy.
5. If a relay signals support for this NIP in their NIP-11 document that means they're willing to accept kind 10002 events from a broad range of users, not only their paying customers or whitelisted group.
6. Clients SHOULD deduplicate connections by normalizing relay URIs according to RFC 3986 .

### Related articles

## NIP-68

## Picture-first feeds

`draft` `optional`

This NIP defines event kind `20` for picture-first clients. Images must be self-contained. They are hosted externally and referenced using `imeta` tags.

The idea is for this type of event to cater to Nostr clients resembling platforms like Instagram, Flickr, Snapchat, or 9GAG, where the picture itself takes center stage in the user experience.

### Picture Events

Picture events contain a `title` tag and description in the `.content`.

They may contain multiple images to be displayed as a single post.

```
{
  "id": <32-bytes lowercase hex-encoded SHA-256 of the the serialized event data>,
  "pubkey": <32-bytes lowercase hex-encoded public key of the event creator>,
  "created_at": <Unix timestamp in seconds>,
  "kind": 20,
  "content": "<description of post>",
  "tags": [
    ["title", "<short title of post>"],

    // Picture Data
    [
      "imeta",
      "url https://nostr.build/i/my-image.jpg",
      "m image/jpeg",
```

```
      "blurhash eVF$^OI:${M{o#*0-nNFxakD-?xVM}WEWB%iNKxvR-oetmo#R-aen$",
      "dim 3024x4032",
      "alt A scenic photo overlooking the coast of Costa Rica",
      "x <sha256 hash as specified in NIP 94>",
      "fallback https://nostrcheck.me/alt1.jpg",
      "fallback https://void.cat/alt1.jpg"
    ],
    [
      "imeta",
      "url https://nostr.build/i/my-image2.jpg",
      "m image/jpeg",
      "blurhash eVF$^OI:${M{o#*0-nNFxakD-?xVM}WEWB%iNKxvR-oetmo#R-aen$",
      "dim 3024x4032",
      "alt Another scenic photo overlooking the coast of Costa Rica",
      "x <sha256 hash as specified in NIP 94>",
      "fallback https://nostrcheck.me/alt2.jpg",
      "fallback https://void.cat/alt2.jpg",

      "annotate-user <32-bytes hex of a pubkey>:<posX>:<posY>" // Tag users in
specific locations in the picture
    ],

    ["content-warning", "<reason>"], // if NSFW

    // Tagged users
    ["p", "<32-bytes hex of a pubkey>", "<optional recommended relay URL>"],
    ["p", "<32-bytes hex of a pubkey>", "<optional recommended relay URL>"],

    // Specify the media type for filters to allow clients to filter by supported
kinds
    ["m", "image/jpeg"],

    // Hashes of each image to make them queryable
    ["x", "<sha256>"]

    // Hashtags
    ["t", "<tag>"],
    ["t", "<tag>"],

    // location
    ["location", "<location>"], // city name, state, country
    ["g", "<geohash>"],

    // When text is written in the image, add the tag to represent the language
    ["L", "ISO-639-1"],
    ["l", "en", "ISO-639-1"]
  ]
}
```

The `imeta` tag `annotate-user` places a user link in the specific position in the image.

Only the following media types are accepted:

- `image/apng`: Animated Portable Network Graphics (APNG)
- `image/avif`: AV1 Image File Format (AVIF)
- `image/gif`: Graphics Interchange Format (GIF)
- `image/jpeg`: Joint Photographic Expert Group image (JPEG)
- `image/png`: Portable Network Graphics (PNG)
- `image/webp`: Web Picture format (WEBP)

Picture events might be used with NIP-71 's kind `34236` to display short vertical videos in the same feed.

## NIP-69

### Peer-to-peer Order events

`draft` `optional`

## Abstract

Peer-to-peer (P2P) platforms have seen an upturn in recent years, while having more and more options is positive, in the specific case of p2p, having several options contributes to the liquidity split, meaning sometimes there's not enough assets available for trading. If we combine all these individual solutions into one big pool of orders, it will make them much more competitive compared to centralized systems, where a single authority controls the liquidity.

This NIP defines a simple standard for peer-to-peer order events, which enables the creation of a big liquidity pool for all p2p platforms participating.

## The event

Events are addressable events and use `38383` as event kind, a p2p event look like this:


```
{
  "id": "84fad0d29cb3529d789faeff2033e88fe157a48e071c6a5d1619928289420e31",
  "pubkey": "dbe0b1be7aafd3cfba92d7463edbd4e33b2969f61bd554d37ac56f032e13355a",
  "created_at": 1702548701,
  "kind": 38383,
  "tags": [
    ["d", "ede61c96-4c13-4519-bf3a-dcf7f1e9d842"],
    ["k", "sell"],
    ["f", "VES"],
    ["s", "pending"],
    ["amt", "0"],
    ["fa", "100"],
    ["pm", "face to face", "bank transfer"],
    ["premium", "1"],
    [
      "rating",
      "
{\"total_reviews\":1,\"total_rating\":3.0,\"last_rating\":3,\"max_rate\":5,\"min_rate\":1}"

    ],
    ["source", "https://t.me/p2plightning/xxxxxxx"],
    ["network", "mainnet"],
    ["layer", "lightning"],
    ["name", "Nakamoto"],
    ["g", "<geohash>"],
    ["bond", "0"],
    ["expiration", "1719391096"],
    ["y", "lnp2pbot"],
    ["z", "order"]
  ],
  "content": "",
  "sig":
"7e8fe1eb644f33ff51d8805c02a0e1a6d034e6234eac50ef7a7e0dac68a0414f7910366204fa8217086f90eddaa37ded71e61

}
```


## Tags

- `d` < Order ID >: A unique identifier for the order.
- `k` < Order type >: `sell` or `buy`.
- `f` < Currency >: The asset being traded, using the ISO 4217 standard.
- `s` < Status >: `pending`, `canceled`, `in-progress`, `success`.
- `amt` < Amount >: The amount of Bitcoin to be traded, the amount is defined in satoshis, if `0` means that the amount of satoshis will be obtained from a public API after the taker accepts the order.
- `fa` < Fiat amount >: The fiat amount being traded, for range orders two values are expected, the minimum and maximum amount.
- `pm` < Payment method >: The payment method used for the trade, if the order has multiple payment methods, they should be separated by a comma.
- `premium` < Premium >: The percentage of the premium the maker is willing to pay.
- `source` [Source]: The source of the order, it can be a URL that redirects to the order.
- `rating` [Rating]: The rating of the maker, this document does not define how the rating is calculated, it's up to the platform to define it.
- `network` < Network >: The network used for the trade, it can be `mainnet`, `testnet`, `signet`, etc.

- `layer` < Layer >: The layer used for the trade, it can be `onchain`, `lightning`, `liquid`, etc.
- `name` [Name]: The name of the maker.
- `g` [Geohash]: The geohash of the operation, it can be useful in a face to face trade.
- `bond` [Bond]: The bond amount, the bond is a security deposit that both parties must pay.
- `expiration` < Expiration>: The expiration date of the order (NIP-40 ).
- `y` < Platform >: The platform that created the order.
- `z` < Document >: `order`.

Mandatory tags are enclosed with `<tag>`, optional tags are enclosed with `[tag]`.

### Implementations

Currently implemented on the following platforms:

- Mostro
- @lnp2pBot
- Robosats

### References

## NIP-70

## Protected Events

`draft` `optional`

When the `"-"` tag is present, that means the event is “protected”.

A protected event is an event that can only be published to relays by its author. This is achieved by relays ensuring that the author is authenticated before publishing their own events or by just rejecting events with `["-"]` outright.

The default behavior of a relay MUST be to reject any event that contains `["-"]`.

Relays that want to accept such events MUST first require that the client perform the NIP-42 `AUTH` flow and then check if the authenticated client has the same pubkey as the event being published and only accept the event in that case.

### The tag

The tag is a simple tag with a single item: `["-"]`. It may be added to any event.

### Example flow

- User `79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798` connects to relay `wss://example.com`:

```
/* client: */
["EVENT",
{"id":"cb8feca582979d91fe90455867b34dbf4d65e4b86e86b3c68c368ca9f9eef6f2","pubkey":"79be667ef9dcbbac55a
[["-"]],"content":"hello members of the secret
group","sig":"fa163f5cfb75d77d9b6269011872ee22b34fb48d23251e9879bb1e4ccbdd8aaaf4b6dc5f5084a65ef42c52fb

/* relay: */
["AUTH", "<challenge>"]
["OK", "cb8feca582979d91fe90455867b34dbf4d65e4b86e86b3c68c368ca9f9eef6f2", false,
"auth-required: this event may only be published by its author"]
/* client: */
["AUTH", {}]
["EVENT",
{"id":"cb8feca582979d91fe90455867b34dbf4d65e4b86e86b3c68c368ca9f9eef6f2","pubkey":"79be667ef9dcbbac55a
[["-"]],"content":"hello members of the secret
group","sig":"fa163f5cfb75d77d9b6269011872ee22b34fb48d23251e9879bb1e4ccbdd8aaaf4b6dc5f5084a65ef42c52fb

["OK", "cb8feca582979d91fe90455867b34dbf4d65e4b86e86b3c68c368ca9f9eef6f2", true, ""]
```

### Why

There are multiple circumstances in which it would be beneficial to prevent the unlimited spreading of an event through all relays imaginable and restrict some to only a certain demographic or to a semi-closed community relay. Even when the information is public it may make sense to keep it compartimentalized across different relays.

It's also possible to create closed access feeds with this when the publisher has some relationship with the relay and trusts the relay to not release their published events to anyone.

Even though it's ultimately impossible to restrict the spread of information on the internet (for example, one of the members of the closed group may want to take an event intended to be restricted and republish it to other relays), most relays would be happy to not facilitate the acts of these so-called "pirates", in respect to the original decision of the author and therefore gladly reject these republish acts if given the means to.

This NIP gives these authors and relays the means to clearly signal when a given event is not intended to be republished by third parties.

## NIP-71

## Video Events

`draft` `optional`

This specification defines video events representing a dedicated post of externally hosted content. These video events are *addressable* and delete-requestable per NIP-09 .

Unlike a `kind 1` event with a video attached, Video Events are meant to contain all additional metadata concerning the subject media and to be surfaced in video-specific clients rather than general micro-blogging clients. The thought is for events of this kind to be referenced in a Netflix, YouTube, or TikTok like nostr client where the video itself is at the center of the experience.

### Video Events

There are two types of video events represented by different kinds: horizontal and vertical video events. This is meant to allow clients to cater to each as the viewing experience for horizontal (landscape) videos is often different than that of vertical (portrait) videos (Stories, Reels, Shorts, etc).

#### Format

The format uses an *addressable event* kind `34235` for horizontal videos and `34236` for vertical videos.

The `.content` of these events is a summary or description on the video content.

The primary source of video information is the `imeta` tags which is defined in NIP-92

Each `imeta` tag can be used to specify a variant of the video by the `dim` & `m` properties.

Example:

```
[
  ["imeta",
    "dim 1920x1080",
    "url https://myvideo.com/1080/12345.mp4",
    "x 3093509d1e0bc604ff60cb9286f4cd7c781553bc8991937befaacfdc28ec5cdc",
    "m video/mp4",
    "image https://myvideo.com/1080/12345.jpg",
    "image https://myotherserver.com/1080/12345.jpg",
    "fallback https://myotherserver.com/1080/12345.mp4",
    "fallback https://andanotherserver.com/1080/12345.mp4",
    "service nip96",
  ],
  ["imeta",
    "dim 1280x720",
    "url https://myvideo.com/720/12345.mp4",
    "x e1d4f808dae475ed32fb23ce52ef8ac82e3cc760702fca10d62d382d2da3697d",
    "m video/mp4",
    "image https://myvideo.com/720/12345.jpg",
    "image https://myotherserver.com/720/12345.jpg",
    "fallback https://myotherserver.com/720/12345.mp4",
    "fallback https://andanotherserver.com/720/12345.mp4",
    "service nip96",
  ],
  ["imeta",
    "dim 1280x720",
    "url https://myvideo.com/720/12345.m3u8",
    "x 704e720af2697f5d6a198ad377789d462054b6e8d790f8a3903afbc1e044014f",
```

```
    "m application/x-mpegURL",
    "image https://myvideo.com/720/12345.jpg",
    "image https://myotherserver.com/720/12345.jpg",
    "fallback https://myotherserver.com/720/12345.m3u8",
    "fallback https://andanotherserver.com/720/12345.m3u8",
    "service nip96",
  ],
]
```

Where `url` is the primary server url and `fallback` are other servers hosting the same file, both `url` and `fallback` should be weighted equally and clients are recommended to use any of the provided video urls.

The `image` tag contains a preview image (at the same resolution). Multiple `image` tags may be used to specify fallback copies in the same way `fallback` is used for `url`.

Additionally `service nip96` may be included to allow clients to search the authors NIP-96 server list to find the file using the hash.

**Other tags:**

- `title` (required) title of the video
- `published_at`, for the timestamp in unix seconds (stringified) of the first time the video was published
- `duration` (optional) video duration in seconds
- `text-track` (optional, repeated) link to WebVTT file for video, type of supplementary information (captions/subtitles/chapters/metadata), optional language code
- `content-warning` (optional) warning about content of NSFW video
- `alt` (optional) description for accessibility
- `segment` (optional, repeated) start timestamp in format `HH:MM:SS.sss`, end timestamp in format `HH:MM:SS.sss`, chapter/segment title, chapter thumbnail-url
- `t` (optional, repeated) hashtag to categorize video
- `p` (optional, repeated) 32-bytes hex pubkey of a participant in the video, optional recommended relay URL
- `r` (optional, repeated) references / links to web pages

```
{
  "id": <32-bytes lowercase hex-encoded SHA-256 of the the serialized event data>,
  "pubkey": <32-bytes lowercase hex-encoded public key of the event creator>,
  "created_at": <Unix timestamp in seconds>,
  "kind": 34235 | 34236,
  "content": "<summary / description of video>",
  "tags": [
    ["d", "<UUID>"],

    ["title", "<title of video>"],
    ["published_at", "<unix timestamp>"],
    ["alt", <description>],

    // Video Data
    ["imeta",
      "dim 1920x1080",
      "url https://myvideo.com/1080/12345.mp4",
      "x 3093509d1e0bc604ff60cb9286f4cd7c781553bc8991937befaacfdc28ec5cdc",
      "m video/mp4",
      "image https://myvideo.com/1080/12345.jpg",
      "image https://myotherserver.com/1080/12345.jpg",
      "fallback https://myotherserver.com/1080/12345.mp4",
      "fallback https://andanotherserver.com/1080/12345.mp4",
      "service nip96",
    ],

    ["duration", <duration of video in seconds>],
    ["text-track", "<encoded `kind 6000` event>", "<recommended relay urls>"],
    ["content-warning", "<reason>"],
    ["segment", <start>, <end>, "<title>", "<thumbnail URL>"],

    // Participants
    ["p", "<32-bytes hex of a pubkey>", "<optional recommended relay URL>"],
    ["p", "<32-bytes hex of a pubkey>", "<optional recommended relay URL>"],
```

```
    // Hashtags
    ["t", "<tag>"],
    ["t", "<tag>"],

    // Reference links
    ["r", "<url>"],
    ["r", "<url>"]
  ]
}
```

## NIP-72

## Moderated Communities (Reddit Style)

`draft` `optional`

The goal of this NIP is to enable public communities. It defines the replaceable event `kind:34550` to define the community and the current list of moderators/administrators. Users that want to post into the community, simply tag any Nostr event with the community's `a` tag. Moderators may issue an approval event `kind:4550`.

## Community Definition

`Kind:34550` SHOULD include any field that helps define the community and the set of moderators. `relay` tags MAY be used to describe the preferred relay to download requests and approvals. A community definition event's `d` tag MAY double as its name, but if a `name` tag is provided, it SHOULD be displayed instead of the `d` tag.

```
{
  "created_at": <Unix timestamp in seconds>,
  "kind": 34550,
  "tags": [
    ["d", "<community-d-identifier>"],
    ["name", "<Community name>"],
    ["description", "<Community description>"],
    ["image", "<Community image url>", "<Width>x<Height>"],

    //.. other tags relevant to defining the community

    // moderators
    ["p", "<32-bytes hex of a pubkey1>", "<optional recommended relay URL>",
"moderator"],
    ["p", "<32-bytes hex of a pubkey2>", "<optional recommended relay URL>",
"moderator"],
    ["p", "<32-bytes hex of a pubkey3>", "<optional recommended relay URL>",
"moderator"],

    // relays used by the community (w/optional marker)
    ["relay", "<relay hosting author kind 0>", "author"],
    ["relay", "<relay where to send and receive requests>", "requests"],
    ["relay", "<relay where to send and receive approvals>", "approvals"],
    ["relay", "<relay where to post requests to and fetch approvals from>"]
  ],
  // other fields...
}
```

## Posting to a community

Any Nostr event can be posted to a community. Clients MUST add one or more community `a` tags, each with a recommended relay.

```
{
  "kind": 1,
  "tags": [
    ["a", "34550:<community event author pubkey>:<community-d-identifier>", "
<optional-relay-url>"],
  ],
  "content": "hello world",
```

```
  // other fields...
}
```

## Moderation

Anyone may issue an approval event to express their opinion that a post is appropriate for a community. Clients MAY choose which approval events to honor, but SHOULD at least use ones published by the group's defined moderators.

An approval event MUST include one or more community `a` tags, an `e` or `a` tag pointing to the post, and the `p` tag of the author of the post (for approval notifications). `a` tag prefixes can be used to disambiguate between community and replaceable event pointers (community `a` tags always begin with `34550`).

The event SHOULD also include the JSON-stringified `post request` event inside the `.content`, and a `k` tag with the original post's event kind to allow filtering of approved posts by kind.

Moderators MAY request deletion of their approval of a post at any time using NIP-09 event deletion requests .

```
{
  "pubkey": "<32-bytes lowercase hex-encoded public key of the event creator>",
  "kind": 4550,
  "tags": [
    ["a", "34550:<event-author-pubkey>:<community-d-identifier>", "<optional-relay-
url>"],
    ["e", "<post-id>", "<optional-relay-url>"],
    ["p", "<port-author-pubkey>", "<optional-relay-url>"],
    ["k", "<post-request-kind>"]
  ],
  "content": "<the full approved event, JSON-encoded>",
  // other fields...
}
```

It's recommended that multiple moderators approve posts to avoid deleting them from the community when a moderator is removed from the owner's list. In case the full list of moderators must be rotated, the new moderator set must sign new approvals for posts in the past or the community will restart. The owner can also periodically copy and re-sign of each moderator's approval events to make sure posts don't disappear with moderators.

Approvals of replaceable events can be created in three ways:

1. By tagging the replaceable event as an `e` tag if moderators want to approve each individual change to the replaceable event
2. By tagging the replaceable event as an `a` tag if the moderator authorizes the replaceable event author to make changes without additional approvals and
3. By tagging the replaceable event with both its `e` and `a` tag which empowers clients to display the original and updated versions of the event, with appropriate remarks in the UI.

Since relays are instructed to delete old versions of a replaceable event, the `content` of an approval using an `e` tag MUST have the specific version of the event or clients might not be able to find that version of the content anywhere.

Clients SHOULD evaluate any non-`34550:* a` tag as posts to be approved for all `34550:* a` tags.

## Cross-posting

Clients MAY support cross-posting between communities by posting a NIP 18 `kind 6` or `kind 16` repost to one or more communities using `a` tags as described above. The `content` of the repost MUST be the original event, not the approval event.

## NIP-73

## External Content IDs

`draft` `optional`

There are certain established global content identifiers such as Book ISBNs , Podcast GUIDs , and Movie ISANs that are useful to reference in nostr events so that clients can query all the events assosiated with these ids.

`i` tags are used for referencing these external content ids, with `k` tags representing the external content id kind so that clients can query all the events for a specific kind.

### Supported IDs

| Type | `i` tag | `k` tag |
|---|---|---|
| URLs | “`<URL, normalized, no fragment>`” | “`<scheme-host, normalized>`” |
| Hashtags | “#`<topic, lowercase>`” | “#” |
| Geohashes | “geo:`<geohash, lowercase>`” | “geo” |
| Books | “isbn:`<id, without hyphens>`” | “isbn” |
| Podcast Feeds | “podcast:guid:`<guid>`” | “podcast:guid” |
| Podcast Episodes | “podcast:item:guid:`<guid>`” | “podcast:item:guid” |
| Podcast Publishers | “podcast:publisher:guid:`<guid>`” | “podcast:publisher:guid” |
| Movies | “isan:`<id, without version part>`” | “isan” |
| Papers | “doi:`<id, lowercase>`” | “doi” |

---

### Examples

**Books:**

- Book ISBN: `["i", "isbn:9780765382030"]` - https://isbnsearch.org/isbn/9780765382030

Book ISBNs MUST be referenced ***without hyphens*** as many book search APIs return the ISBNs without hyphens. Removing hypens from ISBNs is trivial, whereas adding the hyphens back in is non-trivial requiring a library.

**Podcasts:**

- Podcast RSS Feed GUID: `["i", "podcast:guid:c90e609a-df1e-596a-bd5e-57bcc8aad6cc"]` - https://podcastindex.org/podcast/c90e609a-df1e-596a-bd5e-57bcc8aad6cc
- Podcast RSS Item GUID: `["i", "podcast:item:guid:d98d189b-dc7b-45b1-8720-d4b98690f31f"]`
- Podcast RSS Publisher GUID: `["i", "podcast:publisher:guid:18bcbf10-6701-4ffb-b255-bc057390d738"]`

**Movies:**

- Movie ISAN: `["i", "isan:0000-0000-401A-0000-7"]` - https://web.isan.org/public/en/isan/0000-0000-401A-0000-7

Movie ISANs SHOULD be referenced ***without the version part*** as the versions / edits of movies are not relevant. More info on ISAN parts here - https://support.isan.org/hc/en-us/articles/360002783131-Records-relations-and-hierarchies-in-the-ISAN-Registry

---

**Optional URL Hints**

Each `i` tag MAY have a url hint as the second argument to redirect people to a website if the client isn't opinionated about how to interpret the id:

```
["i", "podcast:item:guid:d98d189b-dc7b-45b1-8720-d4b98690f31f",
https://fountain.fm/episode/z1y9TMQRuqXl2awyrQxg]
```

```
["i", "isan:0000-0000-401A-0000-7", https://www.imdb.com/title/tt0120737]
```

## NIP-75

## Zap Goals

`draft` `optional`

This NIP defines an event for creating fundraising goals. Users can contribute funds towards the goal by zapping the goal event.

### Nostr Event

A `kind:9041` event is used.

The `.content` contains a human-readable description of the goal.

The following tags are defined as REQUIRED.

- `amount` - target amount in milisats.
- `relays` - a list of relays the zaps to this goal will be sent to and tallied from.

Example event:

```
{
  "kind": 9041,
  "tags": [
    ["relays", "wss://alicerelay.example.com", "wss://bobrelay.example.com",
/*...*/],
    ["amount", "210000"],
  ],
  "content": "Nostrasia travel expenses",
  // other fields...
}
```

The following tags are OPTIONAL.

- `closed_at` - timestamp for determining which zaps are included in the tally. Zap receipts published after the `closed_at` timestamp SHOULD NOT count towards the goal progress.
- `image` - an image for the goal
- `summary` - a brief description

```
{
  "kind": 9041,
  "tags": [
    ["relays", "wss://alicerelay.example.com", "wss://bobrelay.example.com",
/*...*/],
    ["amount", "210000"],
    ["closed_at", "<unix timestamp in seconds>"],
    ["image", "<image URL>"],
    ["summary", "<description of the goal>"],
  ],
  "content": "Nostrasia travel expenses",
  // other fields...
}
```

The goal MAY include an `r` or `a` tag linking to a URL or addressable event.

The goal MAY include multiple beneficiary pubkeys by specifying `zap` tags .

Addressable events can link to a goal by using a `goal` tag specifying the event id and an optional relay hint.

```
{
  "kind": 3xxxx,
  "tags": [
    ["goal", "<event id>", "<Relay URL (optional)>"],
    // rest of tags...
  ],
  // other fields...
}
```

### Client behavior

Clients MAY display funding goals on user profiles.

When zapping a goal event, clients MUST include the relays in the `relays` tag of the goal event in the zap request `relays` tag.

When zapping an addressable event with a `goal` tag, clients SHOULD tag the goal event id in the `e` tag of the zap request.

### Use cases

- Fundraising clients
- Adding funding goals to events such as long form posts, badges or live streams

## NIP-78

## Arbitrary custom app data

`draft` `optional`

The goal of this NIP is to enable remoteStorage -like capabilities for custom applications that do not care about interoperability.

Even though interoperability is great, some apps do not want or do not need interoperability, and it wouldn't make sense for them. Yet Nostr can still serve as a generalized data storage for these apps in a "bring your own database" way, for example: a user would open an app and somehow input their preferred relay for storage, which would then enable these apps to store application-specific data there.

### Nostr event

This NIP specifies the use of event kind `30078` (an *addressable* event) with a `d` tag containing some reference to the app name and context – or any other arbitrary string. `content` and other `tags` can be anything or in any format.

### Some use cases

- User personal settings on Nostr clients (and other apps unrelated to Nostr)
- A way for client developers to propagate dynamic parameters to users without these having to update
- Personal private data generated by apps that have nothing to do with Nostr, but allow users to use Nostr relays as their personal database

## NIP-84

## Highlights

`draft` `optional`

This NIP defines `kind:9802`, a "highlight" event, to signal content a user finds valuable.

### Format

The `.content` of these events is the highlighted portion of the text.

`.content` might be empty for highlights of non-text based media (e.g. NIP-94 audio/video).

#### References

Events SHOULD tag the source of the highlight, whether nostr-native or not. `a` or `e` tags should be used for nostr events and `r` tags for URLs.

When tagging a URL, clients generating these events SHOULD do a best effort of cleaning the URL from trackers or obvious non-useful information from the query string.

#### Attribution

Clients MAY include one or more `p` tags, tagging the original authors of the material being highlighted; this is particularly useful when highlighting non-nostr content for which the client might be able to get a nostr pubkey somehow (e.g. prompting the user or reading a `<meta name="nostr:nprofile1..." />` tag on the document). A role MAY be included as the last value of the tag.

```
{
  "tags": [
    ["p", "<pubkey-hex>", "<relay-url>", "author"],
    ["p", "<pubkey-hex>", "<relay-url>", "author"],
    ["p", "<pubkey-hex>", "<relay-url>", "editor"]
  ],
  // other fields...
}
```

#### Context

Clients MAY include a `context` tag, useful when the highlight is a subset of a paragraph and displaying the surrounding content might be beneficial to give context to the highlight.

## NIP-86

## Relay Management API

`draft` `optional`

Relays may provide an API for performing management tasks. This is made available as a JSON-RPC-like request-response protocol over HTTP, on the same URI as the relay's websocket.

When a relay receives an HTTP(s) request with a `Content-Type` header of `application/nostr+json+rpc` to a URI supporting WebSocket upgrades, it should parse the request as a JSON document with the following fields:

```
{
  "method": "<method-name>",
  "params": ["<array>", "<of>", "<parameters>"]
}
```

Then it should return a response in the format

```
{
  "result": {"<arbitrary>": "<value>"},
  "error": "<optional error message, if the call has errored>"
}
```

This is the list of **methods** that may be supported:

- `supportedmethods`:
  - params: `[]`
  - result: `["<method-name>", "<method-name>", ...]` (an array with the names of all the other supported methods)
- `banpubkey`:
  - params: `["<32-byte-hex-public-key>", "<optional-reason>"]`
  - result: `true` (a boolean always set to `true`)
- `listbannedpubkeys`:
  - params: `[]`
  - result: `[{"pubkey": "<32-byte-hex>", "reason": "<optional-reason>"}, ...]`, an array of objects
- `allowpubkey`:
  - params: `["<32-byte-hex-public-key>", "<optional-reason>"]`
  - result: `true` (a boolean always set to `true`)
- `listallowedpubkeys`:
  - params: `[]`
  - result: `[{"pubkey": "<32-byte-hex>", "reason": "<optional-reason>"}, ...]`, an array of objects
- `listeventsneedingmoderation`:
  - params: `[]`
  - result: `[{"id": "<32-byte-hex>", "reason": "<optional-reason>"}]`, an array of objects
- `allowevent`:
  - params: `["<32-byte-hex-event-id>", "<optional-reason>"]`
  - result: `true` (a boolean always set to `true`)
- `banevent`:
  - params: `["<32-byte-hex-event-id>", "<optional-reason>"]`
  - result: `true` (a boolean always set to `true`)
- `listbannedevents`:
  - params: `[]`
  - result: `[{"id": "<32-byte hex>", "reason": "<optional-reason>"}, ...]`, an array of objects
- `changerelayname`:
  - params: `["<new-name>"]`
  - result: `true` (a boolean always set to `true`)
- `changerelaydescription`:
  - params: `["<new-description>"]`
  - result: `true` (a boolean always set to `true`)
- `changerelayicon`:
  - params: `["<new-icon-url>"]`
  - result: `true` (a boolean always set to `true`)
- `allowkind`:
  - params: `[<kind-number>]`
  - result: `true` (a boolean always set to `true`)
- `disallowkind`:

    - params: `[<kind-number>]`
    - result: `true` (a boolean always set to `true`)
- `listallowedkinds`:
    - params: `[]`
    - result: `[<kind-number>, ...]`, an array of numbers
- `blockip`:
    - params: `["<ip-address>", "<optional-reason>"]`
    - result: `true` (a boolean always set to `true`)
- `unblockip`:
    - params: `["<ip-address>"]`
    - result: `true` (a boolean always set to `true`)
- `listblockedips`:
    - params: `[]`
    - result: `[{"ip": "<ip-address>", "reason": "<optional-reason>"}, ...]`, an array of objects

#### Authorization

The request must contain an `Authorization` header with a valid NIP-98 event, except the `payload` tag is required. The `u` tag is the relay URL.

If `Authorization` is not provided or is invalid, the endpoint should return a 401 response.

## NIP-89

## Recommended Application Handlers

`draft` `optional`

This NIP describes `kind:31989` and `kind:31990`: a way to discover applications that can handle unknown event-kinds.

### Rationale

Nostr's discoverability and transparent event interaction is one of its most interesting/novel mechanics. This NIP provides a simple way for clients to discover applications that handle events of a specific kind to ensure smooth cross-client and cross-kind interactions.

#### Parties involved

There are three actors to this workflow:

- application that handles a specific event kind (note that an application doesn't necessarily need to be a distinct entity and it could just be the same pubkey as user A)
    - Publishes `kind:31990`, detailing how apps should redirect to it
- user A, who recommends an app that handles a specific event kind
    - Publishes `kind:31989`
- user B, who seeks a recommendation for an app that handles a specific event kind
    - Queries for `kind:31989` and, based on results, queries for `kind:31990`

### Events

#### Recommendation event

```
{
  "kind": 31989,
  "pubkey": <recommender-user-pubkey>,
  "tags": [
    ["d", <supported-event-kind>],
    ["a", "31990:app1-pubkey:<d-identifier>", "wss://relay1", "ios"],
    ["a", "31990:app2-pubkey:<d-identifier>", "wss://relay2", "web"]
  ],
  // other fields...
}
```

The `d` tag in `kind:31989` is the supported event kind this event is recommending.

Multiple `a` tags can appear on the same `kind:31989`.

The second value of the tag SHOULD be a relay hint. The third value of the tag SHOULD be the platform where this recommendation might apply.

### Handler information

```
{
  "kind": 31990,
  "pubkey": "<application-pubkey>",
  "content": "<optional-kind:0-style-metadata>",
  "tags": [
    ["d", <random-id>],
    ["k", <supported-event-kind>],
    ["web", "https://..../a/<bech32>", "nevent"],
    ["web", "https://..../p/<bech32>", "nprofile"],
    ["web", "https://..../e/<bech32>"],
    ["ios", ".../<bech32>"]
  ],
  // other fields...
}
```

- `content` is an optional `metadata`-like stringified JSON object, as described in NIP-01. This content is useful when the pubkey creating the `kind:31990` is not an application. If `content` is empty, the `kind:0` of the pubkey should be used to display application information (e.g. name, picture, web, LUD16, etc.)
- `k` tags' value is the event kind that is supported by this `kind:31990`. Using a `k` tag(s) (instead of having the kind of the `d` tag) provides:
  - Multiple `k` tags can exist in the same event if the application supports more than one event kind and their handler URLs are the same.
  - The same pubkey can have multiple events with different apps that handle the same event kind.
- `bech32` in a URL MUST be replaced by clients with the NIP-19-encoded entity that should be loaded by the application.

Multiple tags might be registered by the app, following NIP-19 nomenclature as the second value of the array.

A tag without a second value in the array SHOULD be considered a generic handler for any NIP-19 entity that is not handled by a different tag.

## Client tag

When publishing events, clients MAY include a `client` tag. Identifying the client that published the note. This tag is a tuple of `name`, `address` identifying a handler event and, a relay `hint` for finding the handler event. This has privacy implications for users, so clients SHOULD allow users to opt-out of using this tag.

```
{
  "kind": 1,
  "tags": [
    ["client", "My Client", "31990:app1-pubkey:<d-identifier>", "wss://relay1"]
  ]
  // other fields...
}
```

### User flow

A user A who uses a non-`kind:1`-centric nostr app could choose to announce/recommend a certain kind-handler application.

When user B sees an unknown event kind, e.g. in a social-media centric nostr client, the client would allow user B to interact with the unknown-kind event (e.g. tapping on it).

The client MIGHT query for the user's and the user's follows handler.

### Example

#### User A recommends a `kind:31337`-handler

User A might be a user of Zapstr, a `kind:31337`-centric client (tracks). Using Zapstr, user A publishes an event recommending Zapstr as a `kind:31337`-handler.

```
{
  "kind": 31989,
```

```
  "tags": [
    ["d", "31337"],
    ["a",
"31990:1743058db7078661b94aaf4286429d97ee5257d14a86d6bfa54cb0482b876fb0:abcd",
<relay-url>, "web"]
  ],
  // other fields...
}
```

#### User B interacts with a `kind:31337`-handler

User B might see in their timeline an event referring to a `kind:31337` event (e.g. a `kind:1` tagging a `kind:31337`).

User B's client, not knowing how to handle a `kind:31337` might display the event using its `alt` tag (as described in NIP-31). When the user clicks on the event, the application queries for a handler for this `kind`:

```
["REQ", <id>, { "kinds": [31989], "#d": ["31337"], "authors": [<user>, <users-
contact-list>] }]
```

User B, who follows User A, sees that `kind:31989` event and fetches the `a`-tagged event for the app and handler information.

User B's client sees the application's `kind:31990` which includes the information to redirect the user to the relevant URL with the desired entity replaced in the URL.

#### Alternative query bypassing `kind:31989`

Alternatively, users might choose to query directly for `kind:31990` for an event kind. Clients SHOULD be careful doing this and use spam-prevention mechanisms or querying high-quality restricted relays to avoid directing users to malicious handlers.

```
["REQ", <id>, { "kinds": [31990], "#k": [<desired-event-kind>], "authors": [...] }]
```

## NIP-90

## Data Vending Machine

`draft` `optional`

This NIP defines the interaction between customers and Service Providers for performing on-demand computation.

Money in, data out.

### Kinds

This NIP reserves the range `5000-7000` for data vending machine use.

| Kind | Description |
|---|---|
| 5000-5999 | Job request kinds |
| 6000-6999 | Job result |
| 7000 | Job feedback |

Job results always use a kind number that is `1000` higher than the job request kind. (e.g. request: `kind:5001` gets a result: `kind:6001`).

Job request types are defined separately .

### Rationale

Nostr can act as a marketplace for data processing, where users request jobs to be processed in certain ways (e.g., "speech-to-text", "summarization", etc.), but they don't necessarily care about "who" processes the data.

This NIP is not to be confused with a 1:1 marketplace; instead, it describes a flow where a user announces a desired output, willingness to pay, and service providers compete to fulfill the job requirement in the best way possible.

#### Actors

There are two actors in the workflow described in this NIP:

- Customers (npubs who request a job)

- Service providers (npubs who fulfill jobs)

### Job request (`kind:5000-5999`)

A request to process data, published by a customer. This event signals that a customer is interested in receiving the result of some kind of compute.

```
{
    "kind": 5xxx, // kind in 5000-5999 range
    "content": "",
    "tags": [
        [ "i", "<data>", "<input-type>", "<relay>", "<marker>" ],
        [ "output", "<mime-type>" ],
        [ "relays", "wss://..." ],
        [ "bid", "<msat-amount>" ],
        [ "t", "bitcoin" ]
    ],
    // other fields...
}
```

All tags are optional.

- `i` tag: Input data for the job (zero or more inputs)
  - `<data>`: The argument for the input
  - `<input-type>`: The way this argument should be interpreted. MUST be one of:
    - `url`: A URL to be fetched of the data that should be processed.
    - `event`: A Nostr event ID.
    - `job`: The output of a previous job with the specified event ID. The dermination of which output to build upon is up to the service provider to decide (e.g. waiting for a signaling from the customer, waiting for a payment, etc.)
    - `text`: `<data>` is the value of the input, no resolution is needed
  - `<relay>`: If `event` or `job` input-type, the relay where the event/job was published, otherwise optional or empty string
  - `<marker>`: An optional field indicating how this input should be used within the context of the job
- `output`: Expected output format. Different job request `kind` defines this more precisely.
- `param`: Optional parameters for the job as key (first argument)/value (second argument). Different job request `kind` defines this more precisely. (e.g. `[ "param", "lang", "es" ]`)
- `bid`: Customer MAY specify a maximum amount (in millisats) they are willing to pay
- `relays`: List of relays where Service Providers SHOULD publish responses to
- `p`: Service Providers the customer is interested in. Other SPs MIGHT still choose to process the job

### Encrypted Params

If the user wants to keep the input parameters a secret, they can encrypt the `i` and `param` tags with the service provider's 'p' tag and add it to the content field. Add a tag `encrypted` as tags. Encryption for private tags will use NIP-04 - Encrypted Direct Message encryption , using the user's private and service provider's public key for the shared secret

```
[
  ["i", "what is the capital of France? ", "text"],
  ["param", "model", "LLaMA-2"],
  ["param", "max_tokens", "512"],
  ["param", "temperature", "0.5"],
  ["param", "top-k", "50"],
  ["param", "top-p", "0.7"],
  ["param", "frequency_penalty", "1"]
]
```

This param data will be encrypted and added to the `content` field and `p` tag should be present

```
{
  "content":
"BE2Y4xvS6HIY7TozIgbEl3sAHkdZoXyLRRkZv4fLPh3R7LtviLKAJM5qpkC7D6VtMbgIt4iNcMpLtpo...",

  "tags": [
    ["p", "04f74530a6ede6b24731b976b8e78fb449ea61f40ff10e3d869a3030c4edc91f"],
    ["encrypted"]
```

```
  ],
  // other fields...
}
```

### Job result (`kind: 6000-6999`)

Service providers publish job results, providing the output of the job result. They should tag the original job request event id as well as the customer's pubkey.

```
{
  "pubkey": "<service-provider pubkey>",
  "content": "<payload>",
  "kind": 6xxx,
  "tags": [
    ["request", "<job-request>"],
    ["e", "<job-request-id>", "<relay-hint>"],
    ["i", "<input-data>"],
    ["p", "<customer's-pubkey>"],
    ["amount", "requested-payment-amount", "<optional-bolt11>"]
  ],
  // other fields...
}
```

- `request`: The job request event stringified-JSON.
- `amount`: millisats that the Service Provider is requesting to be paid. An optional third value can be a bolt11 invoice.
- `i`: The original input(s) specified in the request.

### Encrypted Output

If the request has encrypted params, then output should be encrypted and placed in `content` field. If the output is encrypted, then avoid including `i` tag with input-data as clear text. Add a tag encrypted to mark the output content as `encrypted`

```
{
  "pubkey": "<service-provider pubkey>",
  "content": "<encrypted payload>",
  "kind": 6xxx,
  "tags": [
    ["request", "<job-request>"],
    ["e", "<job-request-id>", "<relay-hint>"],
    ["p", "<customer's-pubkey>"],
    ["amount", "requested-payment-amount", "<optional-bolt11>"],
    ["encrypted"]
  ],
  // other fields...
}
```

### Job feedback

Service providers can give feedback about a job back to the customer.

```
{
  "kind": 7000,
  "content": "<empty-or-payload>",
  "tags": [
    ["status", "<status>", "<extra-info>"],
    ["amount", "requested-payment-amount", "<bolt11>"],
    ["e", "<job-request-id>", "<relay-hint>"],
    ["p", "<customer's-pubkey>"],
  ],
  // other fields...
}
```

- `content`: Either empty or a job-result (e.g. for partial-result samples)
- `amount` tag: as defined in the Job Result section.

- `status` tag: Service Providers SHOULD indicate what this feedback status refers to. Job Feedback Status defines status. Extra human-readable information can be added as an extra argument.
- NOTE: If the input params requires input to be encrypted, then `content` field will have encrypted payload with `p` tag as key.

**Job feedback status**

| status | description |
|---|---|
| `payment-required` | Service Provider requires payment before continuing. |
| `processing` | Service Provider is processing the job. |
| `error` | Service Provider was unable to process the job. |
| `success` | Service Provider successfully processed the job. |
| `partial` | Service Provider partially processed the job. The `.content` might include a sample of the partial results. |

Any job feedback event MIGHT include results in the `.content` field, as described in the Job Result section. This is useful for service providers to provide a sample of the results that have been processed so far.

## Protocol Flow

- Customer publishes a job request (e.g. `kind:5000` speech-to-text).
- Service Providers MAY submit `kind:7000` job-feedback events (e.g. `payment-required`, `processing`, `error`, etc.).
- Upon completion, the service provider publishes the result of the job with a `kind:6000` job-result event.
- At any point, if there is an `amount` pending to be paid as instructed by the service provider, the user can pay the included `bolt11` or zap the job result event the service provider has sent to the user.

Job feedback (`kind:7000`) and Job Results (`kind:6000-6999`) events MAY include an `amount` tag, this can be interpreted as a suggestion to pay. Service Providers MUST use the `payment-required` feedback event to signal that a payment is required and no further actions will be performed until the payment is sent.

Customers can always either pay the included `bolt11` invoice or zap the event requesting the payment and service providers should monitor for both if they choose to include a bolt11 invoice.

### Notes about the protocol flow

The flow is deliberately ambiguous, allowing vast flexibility for the interaction between customers and service providers so that service providers can model their behavior based on their own decisions/perceptions of risk.

Some service providers might choose to submit a `payment-required` as the first reaction before sending a `processing` or before delivering results, some might choose to serve partial results for the job (e.g. a sample), send a `payment-required` to deliver the rest of the results, and some service providers might choose to assess likelihood of payment based on an npub's past behavior and thus serve the job results before requesting payment for the best possible UX.

It's not up to this NIP to define how individual vending machines should choose to run their business.

## Cancellation

A job request might be canceled by publishing a `kind:5` delete request event tagging the job request event.

## Appendix 1: Job chaining

A Customer MAY request multiple jobs to be processed as a chain, where the output of a job is the input of another job. (e.g. podcast transcription -> summarization of the transcription). This is done by specifying as input an event id of a different job with the `job` type.

Service Providers MAY begin processing a subsequent job the moment they see the prior job's result, but they will likely wait for a zap to be published first. This introduces a risk that Service Provider of job #1 might delay publishing the zap event in order to have an advantage. This risk is up to Service Providers to mitigate or to decide whether the service provider of job #1 tends to have good-enough results so as to not wait for an explicit zap to assume the job was accepted.

This gives a higher level of flexibility to service providers (which sophisticated service providers would take anyway).

## Appendix 2: Service provider discoverability

Service Providers MAY use NIP-89 announcements to advertise their support for job kinds:

```
{
  "kind": 31990,
  "pubkey": "<pubkey>",
  "content": "{
    \"name\": \"Translating DVM\",
    \"about\": \"I'm a DVM specialized in translating Bitcoin content.\"
  }",
  "tags": [
    ["k", "5005"], // e.g. translation
    ["t", "bitcoin"] // e.g. optionally advertises it specializes in bitcoin audio
transcription that won't confuse "Drivechains" with "Ridechains"
  ],
  // other fields...
}
```

Customers can use NIP-89 to see what service providers their follows use.

## NIP-92

## Media Attachments

Media attachments (images, videos, and other files) may be added to events by including a URL in the event content, along with a matching `imeta` tag.

`imeta` ("inline metadata") tags add information about media URLs in the event's content. Each `imeta` tag SHOULD match a URL in the event content. Clients may replace imeta URLs with rich previews.

The `imeta` tag is variadic, and each entry is a space-delimited key/value pair. Each `imeta` tag MUST have a `url`, and at least one other field. `imeta` may include any field specified by NIP 94 . There SHOULD be only one `imeta` tag per URL.

### Example

```
{
  "content": "More image metadata tests don't mind me https://nostr.build/i/my-
image.jpg",
  "kind": 1,
  "tags": [
    [
      "imeta",
      "url https://nostr.build/i/my-image.jpg",
      "m image/jpeg",
      "blurhash eVF$^OI:${M{o#*0-nNFxakD-?xVM}WEWB%iNKxvR-oetmo#R-aen$",
      "dim 3024x4032",
      "alt A scenic photo overlooking the coast of Costa Rica",
      "x <sha256 hash as specified in NIP 94>",
      "fallback https://nostrcheck.me/alt1.jpg",
      "fallback https://void.cat/alt1.jpg"
    ]
  ]
}
```

### Recommended client behavior

When uploading files during a new post, clients MAY include this metadata after the file is uploaded and included in the post.

When pasting URLs during post composition, the client MAY download the file and add this metadata before the post is sent.

The client MAY ignore `imeta` tags that do not match the URL in the event content.

## NIP-94

## File Metadata

`draft` `optional`

The purpose of this NIP is to allow an organization and classification of shared files. So that relays can filter and organize in any way that is of interest. With that, multiple types of filesharing clients can be created. NIP-94 support is not expected to be implemented by “social” clients that deal with `kind:1` notes or by longform clients that deal with `kind:30023` articles.

### Event format

This NIP specifies the use of the `1063` event kind, having in `content` a description of the file content, and a list of tags described below:

- `url` the url to download the file
- `m` a string indicating the data type of the file. The MIME types format must be used, and they should be lowercase.
- `x` containing the SHA-256 hexencoded string of the file.
- `ox` containing the SHA-256 hexencoded string of the original file, before any transformations done by the upload server
- `size` (optional) size of file in bytes
- `dim` (optional) size of file in pixels in the form `<width>x<height>`
- `magnet` (optional) URI to magnet file
- `i` (optional) torrent infohash
- `blurhash`(optional) the blurhash to show while the file is being loaded by the client
- `thumb` (optional) url of thumbnail with same aspect ratio
- `image` (optional) url of preview image with same dimensions
- `summary` (optional) text excerpt
- `alt` (optional) description for accessibility
- `fallback` (optional) zero or more fallback file sources in case `url` fails
- `service` (optional) service type which is serving the file (eg. NIP-96 )

```
{
  "kind": 1063,
  "tags": [
    ["url",<string with URI of file>],
    ["m", <MIME type>],
    ["x", <Hash SHA-256>],
    ["ox", <Hash SHA-256>],
    ["size", <size of file in bytes>],
    ["dim", <size of file in pixels>],
    ["magnet", <magnet URI> ],
    ["i", <torrent infohash>],
    ["blurhash", <value>],
    ["thumb", <string with thumbnail URI>, <Hash SHA-256>],
    ["image", <string with preview URI>, <Hash SHA-256>],
    ["summary", <excerpt>],
    ["alt", <description>]
  ],
  "content": "<caption>",
  // other fields...
}
```

### Suggested use cases

- A relay for indexing shared files. For example, to promote torrents.
- A pinterest-like client where people can share their portfolio and inspire others.
- A simple way to distribute configurations and software updates.

## NIP-96

## HTTP File Storage Integration

`draft` `optional`

### Introduction

This NIP defines a REST API for HTTP file storage servers intended to be used in conjunction with the nostr network. The API will enable nostr users to upload files and later reference them by url on nostr notes.

The spec DOES NOT use regular nostr events through websockets for storing, requesting nor retrieving data because, for simplicity, the server will not have to learn anything about nostr relays.

### Server Adaptation

File storage servers wishing to be accessible by nostr users should opt-in by making available an https route at `/.well-known/nostr/nip96.json` with `api_url`:

```
{
  // Required
  // File upload and deletion are served from this url
  // Also downloads if "download_url" field is absent or empty string
  "api_url": "https://your-file-server.example/custom-api-path",
  // Optional
  // If absent, downloads are served from the api_url
  "download_url": "https://a-cdn.example/a-path",
  // Optional
  // Note: This field is not meant to be set by HTTP Servers.
  // Use this if you are a nostr relay using your /.well-known/nostr/nip96.json
  // just to redirect to someone else's http file storage server's /.well-
known/nostr/nip96.json
  // In this case, "api_url" field must be an empty string
  "delegated_to_url": "https://your-file-server.example",
  // Optional
  "supported_nips": [60],
  // Optional
  "tos_url": "https://your-file-server.example/terms-of-service",
  // Optional
  "content_types": ["image/jpeg", "video/webm", "audio/*"],
  // Optional
  "plans": {
    // "free" is the only standardized plan key and
    // clients may use its presence to learn if server offers free storage
    "free": {
      "name": "Free Tier",
      // Default is true
      // All plans MUST support NIP-98 uploads
      // but some plans may also allow uploads without it
      "is_nip98_required": true,
      "url": "https://...", // plan's landing page if there is one
      "max_byte_size": 10485760,
      // Range in days / 0 for no expiration
      // [7, 0] means it may vary from 7 days to unlimited persistence,
      // [0, 0] means it has no expiration
      // early expiration may be due to low traffic or any other factor
      "file_expiration": [14, 90],
      "media_transformations": {
        "image": [
          "resizing"
        ]
      }
    }
  }
}
```

#### Relay Hints

Note: This section is not meant to be used by HTTP Servers.

A nostr relay MAY redirect to someone else's HTTP file storage server by adding a `/.well-known/nostr/nip96.json` with "delegated_to_url" field pointing to the url where the server hosts its own `/.well-known/nostr/nip96.json`. In this case, the "api_url" field must be an empty string and all other fields must be absent.

If the nostr relay is also an HTTP file storage server, it must use the "api_url" field instead.

#### List of Supporting File Storage Servers

See https://github.com/aljazceru/awesome-nostr#nip-96-file-storage-servers .

### Auth

When indicated, `clients` must add an NIP-98 `Authorization` header (**optionally** with the encoded `payload` tag set to the base64-encoded 256-bit SHA-256 hash of the file - not the hash of the whole request body).

### Upload

`POST $api_url` as `multipart/form-data`.

#### AUTH required

List of form fields:

- `file`: **REQUIRED** the file to upload
- `caption`: **RECOMMENDED** loose description;
- `expiration`: UNIX timestamp in seconds. Empty string if file should be stored forever. The server isn't required to honor this.
- `size`: File byte size. This is just a value the server can use to reject early if the file size exceeds the server limits.
- `alt`: **RECOMMENDED** strict description text for visibility-impaired users.
- `media_type`: "avatar" or "banner". Informs the server if the file will be used as an avatar or banner. If absent, the server will interpret it as a normal upload, without special treatment.
- `content_type`: mime type such as "image/jpeg". This is just a value the server can use to reject early if the mime type isn't supported.
- `no_transform`: "true" asks server not to transform the file and serve the uploaded file as is, may be rejected.

Others custom form data fields may be used depending on specific `server` support. The `server` isn't required to store any metadata sent by `clients`.

The `filename` embedded in the file may not be honored by the `server`, which could internally store just the SHA-256 hash value as the file name, ignoring extra metadata. The hash is enough to uniquely identify a file, that's why it will be used on the `download` and `delete` routes.

The `server` MUST link the user's `pubkey` string as the owner of the file so to later allow them to delete the file.

`no_transform` can be used to replicate a file to multiple servers for redundancy, clients can use the server list to find alternative servers which might contain the same file. When uploading a file and requesting `no_transform` clients should check that the hash matches in the response in order to detect if the file was modified.

#### Response codes

- `200 OK`: File upload exists, but is successful (Existing hash)
- `201 Created`: File upload successful (New hash)
- `202 Accepted`: File upload is awaiting processing, see Delayed Processing section
- `413 Payload Too Large`: File size exceeds limit
- `400 Bad Request`: Form data is invalid or not supported.
- `403 Forbidden`: User is not allowed to upload or the uploaded file hash didnt match the hash included in the `Authorization` header `payload` tag.
- `402 Payment Required`: Payment is required by the server, **this flow is undefined**.

The upload response is a json object as follows:

```
{
  // "success" if successful or "error" if not
  "status": "success",
  // Free text success, failure or info message
  "message": "Upload successful.",
  // Optional. See "Delayed Processing" section
  "processing_url": "...",
  // This uses the NIP-94 event format but DO NOT need
  // to fill some fields like "id", "pubkey", "created_at" and "sig"
  //
  // This holds the download url ("url"),
  // the ORIGINAL file hash before server transformations ("ox")
  // and, optionally, all file metadata the server wants to make available
  //
  // nip94_event field is absent if unsuccessful upload
```

```
  "nip94_event": {
    // Required tags: "url" and "ox"
    "tags": [
      // Can be same from /.well-known/nostr/nip96.json's "download_url" field
      // (or "api_url" field if "download_url" is absent or empty) with appended
      // original file hash.
      //
      // Note we appended .png file extension to the `ox` value
      // (it is optional but extremely recommended to add the extension as it will
help nostr clients
      // with detecting the file type by using regular expression)
      //
      // Could also be any url to download the file
      // (using or not using the /.well-known/nostr/nip96.json's "download_url"
prefix),
      // for load balancing purposes for example.
      ["url", "https://your-file-server.example/custom-api-
path/719171db19525d9d08dd69cb716a18158a249b7b3b3ec4bbdec5698dca104b7b.png"],
      // SHA-256 hash of the ORIGINAL file, before transformations.
      // The server MUST store it even though it represents the ORIGINAL file
because
      // users may try to download/delete the transformed file using this value
      ["ox", "719171db19525d9d08dd69cb716a18158a249b7b3b3ec4bbdec5698dca104b7b"],
      // Optional. SHA-256 hash of the saved file after any server transformations.
      // The server can but does not need to store this value.
      ["x", "543244319525d9d08dd69cb716a18158a249b7b3b3ec4bbde5435543acb34443"],
      // Optional. Recommended for helping clients to easily know file type before
downloading it.
      ["m", "image/png"],
      // Optional. Recommended for helping clients to reserve an adequate UI space
to show the file before downloading it.
      ["dim", "800x600"]
      // ... other optional NIP-94 tags
    ],
    "content": ""
  },
  // ... other custom fields (please consider adding them to this NIP or to NIP-94
tags)
}
```

Note that if the server didn't apply any transformation to the received file, both `nip94_event.tags.*.ox` and `nip94_event.tags.*.x` fields will have the same value. The server MUST link the saved file to the SHA-256 hash of the **original** file before any server transformations (the `nip94_event.tags.*.ox` tag value). The **original** file's SHA-256 hash will be used to identify the saved file when downloading or deleting it.

`clients` may upload the same file to one or many `servers`. After successful upload, the `client` may optionally generate and send to any set of nostr `relays` a NIP-94 event by including the missing fields.

Alternatively, instead of using NIP-94, the `client` can share or embed on a nostr note just the above url.

`clients` may also use the tags from the `nip94_event` to construct an `imeta` tag

**Delayed Processing**

Sometimes the server may want to place the uploaded file in a processing queue for deferred file processing.

In that case, the server MUST serve the original file while the processing isn't done, then swap the original file for the processed one when the processing is over. The upload response is the same as usual but some optional metadata like `nip94_event.tags.*.x` and `nip94_event.tags.*.size` won't be available.

The expected resulting metadata that is known in advance should be returned on the response. For example, if the file processing would change a file from "jpg" to "webp", use ".webp" extension on the `nip94_event.tags.*.url` field value and set "image/webp" to the `nip94_event.tags.*.m` field. If some metadata are unknown before processing ends, omit them from the response.

The upload response MAY include a `processing_url` field informing a temporary url that may be used by clients to check if the file processing is done.

If the processing isn't done, the server should reply at the `processing_url` url with **200 OK** and the following JSON:

```
{
  // It should be "processing". If "error" it would mean the processing failed.
  "status": "processing",
  "message": "Processing. Please check again later for updated status.",
  "percentage": 15 // Processing percentage. An integer between 0 and 100.
}
```

When the processing is over, the server replies at the `processing_url` url with **201 Created** status and a regular successful JSON response already mentioned before (now **without** a `processing_url` field), possibly including optional metadata at `nip94_event.tags.*` fields that weren't available before processing.

#### File compression

File compression and other transformations like metadata stripping can be applied by the server. However, for all file actions, such as download and deletion, the **original** file SHA-256 hash is what identifies the file in the url string.

### Download

```
GET $api_url/<sha256-hash>(.ext)
```

The primary file download url informed at the upload's response field `nip94_event.tags.*.url` can be that or not (it can be any non-standard url the server wants). If not, the server still MUST also respond to downloads at the standard url mentioned on the previous paragraph, to make it possible for a client to try downloading a file on any NIP-96 compatible server by knowing just the SHA-256 file hash.

Note that the "<sha256-hash>" part is from the **original** file, **not** from the **transformed** file if the uploaded file went through any server transformation.

Supporting ".ext", meaning "file extension", is required for `servers`. It is optional, although recommended, for `clients` to append it to the path. When present it may be used by `servers` to know which `Content-Type` header to send (e.g.: "Content-Type": "image/png" for ".png" extension). The file extension may be absent because the hash is the only needed string to uniquely identify a file.

Example: `$api_url/719171db19525d9d08dd69cb716a18158a249b7b3b3ec4bbdec5698dca104b7b.png`

#### Media Transformations

`servers` may respond to some media transformation query parameters and ignore those they don't support by serving the original media file without transformations.

##### Image Transformations

###### Resizing

Upon upload, `servers` may create resized image variants, such as thumbnails, respecting the original aspect ratio. `clients` may use the `w` query parameter to request an image version with the desired pixel width. `servers` can then serve the variant with the closest width to the parameter value or an image variant generated on the fly.

Example: `$api_url/<sha256-hash>.png?w=32`

### Deletion

```
DELETE $api_url/<sha256-hash>(.ext)
```

#### AUTH required

Note that the `/<sha256-hash>` part is from the **original** file, **not** from the **transformed** file if the uploaded file went through any server transformation.

The extension is optional as the file hash is the only needed file identification.

The `server` should reject deletes from users other than the original uploader with the appropriate http response code (403 Forbidden).

It should be noted that more than one user may have uploaded the same file (with the same hash). In this case, a delete must not really delete the file but just remove the user's `pubkey` from the file owners list (considering the server keeps just one copy of the same file, because multiple uploads of the same file results in the same file hash).

The successful response is a 200 OK one with just basic JSON fields:

```
{
  "status": "success",
  "message": "File deleted."
}
```

### Listing files

`GET $api_url?page=x&count=y`

**AUTH required**

Returns a list of files linked to the authenticated users pubkey.

Example Response:

```
{
  "count": 1, // server page size, eg. max(1, min(server_max_page_size, arg_count))
  "total": 1, // total number of files
  "page": 0, // the current page number
  "files": [
    {
      "tags": [
        ["ox", "719171db19525d9d08dd69cb716a18158a249b7b3b3ec4bbdec5698dca104b7b"],
        ["x", "5d2899290e0e69bcd809949ee516a4a1597205390878f780c098707a7f18e3df"],
        ["size", "123456"],
        ["alt", "a meme that makes you laugh"],
        ["expiration",  "1715691139"],
        // ...other metadata
      ],
      "content": "haha funny meme", // caption
      "created_at": 1715691130 // upload timestamp
    }
  ]
}
```

`files` contains an array of NIP-94 events

**Query args**

- `page` page number (`offset=page*count`)
- `count` number of items per page

### Selecting a Server

Note: HTTP File Storage Server developers may skip this section. This is meant for client developers.

A File Server Preference event is a kind 10096 replaceable event meant to select one or more servers the user wants to upload files to. Servers are listed as `server` tags:

```
{
  "kind": 10096,
  "content": "",
  "tags": [
    ["server", "https://file.server.one"],
    ["server", "https://file.server.two"]
  ],
  // other fields...
}
```

## NIP-98

## HTTP Auth

`draft` `optional`

This NIP defines an ephemeral event used to authorize requests to HTTP servers using nostr events.

This is useful for HTTP services which are built for Nostr and deal with Nostr user accounts.

### Nostr event

A `kind 27235` (In reference to RFC 7235 ) event is used.

The `content` SHOULD be empty.

The following tags MUST be included.

- `u` - absolute URL
- `method` - HTTP Request Method

Example event:

```
{
  "id": "fe964e758903360f28d8424d092da8494ed207cba823110be3a57dfe4b578734",
  "pubkey": "63fe6318dc58583cfe16810f86dd09e18bfd76aabc24a0081ce2856f330504ed",
  "content": "",
  "kind": 27235,
  "created_at": 1682327852,
  "tags": [
    ["u", "https://api.snort.social/api/v1/n5sp/list"],
    ["method", "GET"]
  ],
  "sig":
"5ed9d8ec958bc854f997bdc24ac337d005af372324747efe4a00e24f4c30437ff4dd8308684bed467d9d6be3e5a517bb43b17

}
```

Servers MUST perform the following checks in order to validate the event:

1. The `kind` MUST be `27235`.
2. The `created_at` timestamp MUST be within a reasonable time window (suggestion 60 seconds).
3. The `u` tag MUST be exactly the same as the absolute request URL (including query parameters).
4. The `method` tag MUST be the same HTTP method used for the requested resource.

When the request contains a body (as in POST/PUT/PATCH methods) clients SHOULD include a SHA256 hash of the request body in a `payload` tag as hex (`["payload", "<sha256-hex>"]`), servers MAY check this to validate that the requested payload is authorized.

If one of the checks was to fail the server SHOULD respond with a 401 Unauthorized response code.

Servers MAY perform additional implementation-specific validation checks.

### Request Flow

Using the `Authorization` HTTP header, the `kind 27235` event MUST be `base64` encoded and use the Authorization scheme `Nostr`

Example HTTP Authorization header:

```
Authorization: Nostr
eyJpZCI6ImZlOTY0ZTc1ODkwMzM2MGYyOGQ4NDI0ZDA5MmRhODQ5NGVkMjA3Y2JhODIzMTEwYmUzYTU3ZGZlNGI1Nzg3MzQiLCJwdW
```

### Reference Implementations

- C# ASP.NET `AuthenticationHandler` NostrAuth.cs

## NIP-99

## Classified Listings

`draft` `optional`

This NIP defines `kind:30402`: an addressable event to describe classified listings that list any arbitrary product, service, or other thing for sale or offer and includes enough structured metadata to make them useful.

The category of classifieds includes a very broad range of physical goods, services, work opportunities, rentals, free giveaways, personals, etc. and is distinct from the more strictly structured marketplaces defined in NIP-15 that often sell many units of specific products through very specific channels.

The structure of these events is very similar to NIP-23 long-form content events.

**Draft / Inactive Listings**

`kind:30403` has the same structure as `kind:30402` and is used to save draft or inactive classified listings.

**Content**

The `.content` field should be a description of what is being offered and by whom. These events should be a string in Markdown syntax.

**Author**

The `.pubkey` field of these events are treated as the party creating the listing.

**Metadata**

- For “tags”/“hashtags” (i.e. categories or keywords of relevance for the listing) the `"t"` event tag should be used.
- For images, whether included in the markdown content or not, clients SHOULD use `image` tags as described in NIP-58 . This allows clients to display images in carousel format more easily.

The following tags, used for structured metadata, are standardized and SHOULD be included. Other tags may be added as necessary.

- `"title"`, a title for the listing
- `"summary"`, for short tagline or summary for the listing
- `"published_at"`, for the timestamp (in unix seconds – converted to string) of the first time the listing was published.
- `"location"`, for the location.
- `"price"`, for the price of the thing being listed. This is an array in the format `[ "price", "<number>", "<currency>", "<frequency>" ]`.
  - `"price"` is the name of the tag
  - `"<number>"` is the amount in numeric format (but included in the tag as a string)
  - `"<currency>"` is the currency unit in 3-character ISO 4217 format or ISO 4217-like currency code (e.g. `"btc"`, `"eth"`).
  - `"<frequency>"` is optional and can be used to describe recurring payments. SHOULD be in noun format (hour, day, week, month, year, etc.)
- 
  - `"status"` (optional), the status of the listing. SHOULD be either “active” or “sold”.

**`price` examples**

- $50 one-time payment `["price", "50", "USD"]`
- €15 per month `["price", "15", "EUR", "month"]`
- £50,000 per year `["price", "50000", "GBP", "year"]`

Other standard tags that might be useful.

- `"g"`, a geohash for more precise location

## Example Event

```
{
  "kind": 30402,
  "created_at": 1675642635,
  // Markdown content
  "content": "Lorem [ipsum]
[nostr:nevent1qqst8cujky046negxgwwm5ynqwn53t8aqjr6afd8g59nfqwxpdhylpcpzamhxue69uhhyetvv9ujuetcv9khqmr9
 dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut
labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation
ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in
reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.
Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt
mollit anim id est laborum.\n\nRead more at
nostr:naddr1qqzkjurnw4ksz9thwden5te0wfjkccte9ehx7um5wghx7un8qgs2d90kkcq3nk2jry62dyf50k0h36rhpdtd594my4

  "tags": [
    ["d", "lorem-ipsum"],
```

```
    ["title", "Lorem Ipsum"],
    ["published_at", "1296962229"],
    ["t", "electronics"],
    ["image", "https://url.to.img", "256x256"],
    ["summary", "More lorem ipsum that is a little more than the title"],
    ["location", "NYC"],
    ["price", "100", "USD"],
    [
      "e",
      "b3e392b11f5d4f28321cedd09303a748acfd0487aea5a7450b3481c60b6e4f87",
      "wss://relay.example.com"
    ],
    [
      "a",

"30023:a695f6b60119d9521934a691347d9f78e8770b56da16bb255ee286ddf9fda919:ipsum",
      "wss://relay.nostr.org"
    ]
  ],
  "pubkey": "...",
  "id": "..."
}
```

## References

- https://github.com/nostr-protocol/nips
- https://github.com/nostr-protocol/nostr

---

Source: nostr-protocol/nips version: 42370a3 2024-12-31T15:38:35Z