Nostr Book of NIPs

TODO

Abstract

This is the abstract.
It consists of two paragraphs.

Contents

Introduction

Welcome to the nostr-book, which is a streamlined guide to the Nostr Notes in Progress (NIPs). Instead of sticking to the original numerical order, I've grouped similar NIPs together to make them easier to understand and more practical to use. Whether you're new to Nostr or a seasoned participant, this reorganized format should help you get a better grip on how things work and what's being developed.

I want to be clear: I didn't write the NIPs. All the credit goes to the original authors and contributors of these notes. My contribution has been to sort these NIPs into a flow that makes sense and brings out the connections between them, making everything more accessible.

Each section of this book kicks off with a short introduction to give you a heads-up on what to expect from the NIPs that follow. The goal is to make the technical details a bit friendlier and the big ideas a bit clearer, so more people can join in, understand, and contribute to the Nostr community.

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

Nostr Overview

…

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: - No whitespace, line breaks or other unnecessary formatting should be included in the output JSON. - No characters except the following should be escaped, and instead should be included verbatim: - A line break, `0x0A`, as `\n` - A double quote, `0x22`, as `\"` - A backslash, `0x5C`, as `\\` - A carriage return, `0x0D`, as `\r` - A tab character, `0x09`, as `\t` - A backspace, `0x08`, as `\b` - A form feed, `0x0C`, as `\f` - UTF-8 should be used for encoding.

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 a (maybe parameterized) replaceable event
  - for a parameterized replaceable event: `["a", <kind integer>:<32-bytes lowercase hex of a pubkey>:<d tag value>, <recommended relay URL, optional>]`
  - for a non-parameterized 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.

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`: **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`, 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 **parameterized replaceable**, which means that, for each combination of `pubkey`, `kind` and the `d` tag's first 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 be newer than this to pass>,
  "until": <an integer unix timestamp in seconds, events must be older than this to pass>,
  "limit": <maximum number of events relays SHOULD return in the initial query>
}
```

Upon receiving a `REQ` message, the relay SHOULD query its internal database and return events that match the filter, then store that filter and send again all future events it receives to that same websocket until the websocket is closed. The `CLOSE` event is received with the same `<subscription_id>` or a new `REQ` is sent using the same `<subscription_id>`, in which case relay MUST overwrite the previous subscription.

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`. 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.`
  - `["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", "duplicate: sub1 already opened"]`
  - `["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.

Communication

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.

Positional “e” tags (DEPRECATED)

> This scheme is in common use; but should be considered deprecated.

`["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.

  > This scheme is deprecated because it creates ambiguities that are difficult, or impossible to resolve when an event references another but is not a reply.

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.

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-23

Long-form Content

`draft` `optional`

This NIP defines `kind:30023` (a *parameterized replaceable 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, as per NIP-12.

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 make use of the parameterized replaceability feature and 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:nevent1qqst8cujky046negxgwwm5ynqwn53t8aqjr6afd8g59nfqwxpdhylpcpzamhxue69uhhyetvv9ujuetcv9khqmr99e3k7mg8arnc9]
      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:naddr1qqzkjurnw4ksz9thwden5te0wfjkccte9ehx7um5wghx7un8qgs2d90kkcq3nk2jry62dyf50k0h36rhpdtd594my40w9pkal876jxgrqsqqqa2
```

```
  "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 defines extra optional fields added to events.

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 contacts 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
- `title`: name of NIP-51 sets, NIP-52 calendar event, NIP-53 live event or NIP-99 listing

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": now(),
  "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>"],
    ...
  ],
  "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 14s 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:14` 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 `nsec1w8udu59ydjvedgs3yv5qccshcj8k05fh3l60k9x57asjrqdpa00qkmr89` to `nsec12ywtkplvyq5t6twdqwwygavp5lm4fhuang89c943nf2z92eez43szvn4dt`.

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


```
{
  "id":"2886780f7349afc1344047524540ee716f7bdc1b64191699855662330bf235d8",
  "pubkey":"8f8a7ec43b77d25799281207e1a47f7a654755055788f7482653f9c9661c6d51",
  "created_at":1703128320,
  "kind":1059,
  "tags":[
    [ "p", "918e2da906df4ccd12c8ac672d8335add131a4cf9d27ce42b3bb3625755f0788"]
  ],
  "content":"AsqzdlMsG304G8h08bE67dhAR1gFTzTckUUyuvndZ8LrGCvwI4pgC3d6hyAK0Wo9gtkLqSr2rT2RyHlE5wRqbC0lQ8WvJEKwqwIJwT5P03l2RxvGCHDb
  "sig":"a3c6ce632b145c0869423c1afaff4a6d764a9b64dedaf15f170b944ead67227518a72e455567ca1c2a0d187832cecbde7ed478395ec4c95dd3e7149
}
```

```
{
  "id":"162b0611a1911cfcb30f8a5502792b346e535a45658b3a31ae5c178465509721",
  "pubkey":"626be2af274b29ea4816ad672ee452b7cf96bbb4836815a55699ae402183f512",
  "created_at":1702711587,
  "kind":1059,
  "tags":[
    [ "p", "449000586091b284416a0c001f677f9c49f7639a55c3f1e2ec130a8e1a7998e1b"]
  ],
  "content":"AsTClTzr0gzXXji7uye5UB6LYrx3HDjWGdkNaBS6BAX9CpHa+Vvtt5oI2xJrmWLen+Fo2NBOFazvl285Gb3HSM82gVycrzx1HUAaQDUG6HI7XBEGqBhQ
  "sig":"c94e74533b482aa8eeeb54ae72a5303e0b21f62909ca43c8ef06b0357412d6f8a92f96e1a205102753777fd25321a58fba3fb384eee114bd53ce6c06
}
```

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-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.

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

```
{
  "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-09

Event Deletion

`draft` `optional`

A special event with kind 5, meaning "deletion" is defined as having a list of one or more `e` tags, each referencing an event the author is requesting to be deleted.

Each tag entry must contain an "e" event id and/or `a` tags intended for deletion.

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

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>"]
  ],
  "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 status for referenced events.

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

Client Usage

Clients MAY choose to fully hide any events that are referenced by valid deletion 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 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 event itself in any way they choose, e.g., not at all, or with a prominent notice.

Relay Usage

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

Deleting a Deletion

Publishing a deletion event against a deletion has no effect. Clients and relays are not obliged to support "undelete" functionality.

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 'dont 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.

Social

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 `f57f54057d2a7af0efecc8b0b66f` the client can show *frank.david.erin* instead.

NIP-05

Mapping Nostr keys to DNS-based internet identifiers

`final` `optional`

On events of kind `0` (`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 `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\"}"
  ...
}
```

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

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.

User Discovery implementation suggestion

A client can also 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.

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-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 `a`, `e` and `p` tags pointing to the note the user is reacting to. The `p` tag allows authors to be notified. The `e` tags enables clients to pull all the reactions to individual events and `a` tags enables clients to seek reactions for all versions of a replaceable event.

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

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

The `p` tag MUST be the `pubkey` of the event 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 {
    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
}
```

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-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
}
```

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.

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-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":
      "f8c8bab1b90cc3d2ae1ad999e6af8af449ad8bb4edf64807386493163e29162b5852a796a8f474d6b1001cddbaac0de4392838574f5366f03cc94cf5df
  "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.

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 `27866e9d854c78ae625b867eefdfa9580434bc3e675be08d2acb526610d96fb` 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-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 parameterized replaceable 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 parameterized replaceable 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-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 a parameterized replaceable 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 a parameterized replaceable 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 `widthxheight` 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 `widthxheight` 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"]
  ],
  ...
}
```

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"]
  ],
  ...
}
```

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"]
  ],
  ...
}
```

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 contents in addition to name, about, picture fields as included in NIP-01:

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

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>`.

Groups

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\"]}",
  ...
}
```

Kind 41: Set channel metadata

Update a channel's public metadata.

Clients and relays SHOULD handle kind 41 events similar to kind 33 replaceable events, where the information is used to update the metadata, without modifying the event id for the channel.Only the most recent kind 41 is needed to be stored.

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>]],
  ...
}
```

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"]],
  ...
}
```

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>],
      ...
  ],
  ...
}
```

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>]],
  ...
}
```

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>]],
  ...
```

```
}
```

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 parameterized replaceable 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`.

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.

Event definitions

- *text root note* (`kind:11`)

This is the basic unit of a "microblog" root text note sent to a group.

```
"kind": 11,
"content": "hello my friends lovers of pizza",
"tags": [
  ["h", "<group-id>"],
  ["previous", "<event-id-first-chars>", "<event-id-first-chars>", ...]
```

```
  ]
  ...
```

- *threaded text reply* (`kind:12`)

This is the basic unit of a "microblog" reply note sent to a group. It's the same as `kind:11`, except for the fact that it must be used whenever it's in reply to some other note (either in reply to a `kind:11` or a `kind:12`). `kind:12` events SHOULD use NIP-10 markers, leaving an empty relay url:

- `["e", "<kind-11-root-id>", "", "root"]`
- `["e", "<kind-12-event-id>", "", "reply"]`
- *chat message* (`kind:9`)

This is the basic unit of a *chat message* sent to a group.

```
  "kind": 9,
  "content": "hello my friends lovers of pizza",
  "tags": [
    ["h", "<group-id>"],
    ["previous", "<event-id-first-chars>", "<event-id-first-chars>", ...]
  ]
  ...
```

- *chat message threaded reply* (`kind:10`)

Similar to `kind:12`, this is the basic unit of a chat message sent to a group. This is intended for in-chat threads that may be hidden by default. Not all in-chat replies MUST use `kind:10`, only when the intention is to create a hidden thread that isn't part of the normal flow of the chat (although clients are free to display those by default too).

`kind:10` SHOULD use NIP-10 markers, just like `kind:12`.

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

Any user can send one of these events to the relay in order to be automatically or manually added to the group. If the group is `open` the relay will automatically issue a `kind:9000` in response adding this user. Otherwise group admins may choose to query for these requests and act upon them.

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

- *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. The relay may discard the event after taking action or keep it as a moderation log.

```
{
  "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	`add-user`	`p` (pubkey hex)
9001	`remove-user`	`p` (pubkey hex)
9002	`edit-metadata`	`name`, `about`, `picture` (string)
9003	`add-permission`	`p` (pubkey), `permission` (name)
9004	`remove-permission`	`p` (pubkey), `permission` (name)
9005	`delete-event`	`e` (id hex)
9006	`edit-group-status`	`public` or `private`, `open` or `closed`

- *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.

```
{
  "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"]
  ]
  ...
}
```

`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)

Similar to the group metadata, this event is supposed to be generated by relays that host the group.

Each admin gets a label that is only used for display purposes, and a list of permissions it has are listed afterwards. These permissions can inform client building UI, but ultimately are evaluated by the relay in order to become effective.

The list of capabilities, as defined by this NIP, for now, is the following:

- `add-user`
- `edit-metadata`
- `delete-event`
- `remove-user`
- `add-permission`
- `remove-permission`
- `edit-group-status`

```
{
  "kind": 39001,
  "content": "list of admins for the pizza lovers group",
  "tags": [
    ["d", "<group-id>"],
    ["p", "<pubkey1-as-hex>", "ceo", "add-user", "edit-metadata", "delete-event", "remove-user"],
    ["p", "<pubkey2-as-hex>", "secretary", "add-user", "delete-event"]
```

```
  ]
  ...
}
```

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

Similar to *group admins,* this event is supposed to be generated by relays that host the group.

It's a NIP-51-like list of pubkeys that are members of the group. Relays might choose to not to publish this information or to restrict what pubkeys can fetch it.

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

Storing the list of groups a user belongs to

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.

Moderation

NIP-32

Labeling

`draft` `optional`

A label is a `kind 1985` event that is used to label other entities. This supports a number of use cases, including distributed moderation, collection management, license assignment, and content classification.

This NIP introduces two 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 REQUIRED 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. `l` tags MUST include a `mark` matching an `L` tag value in the same event.

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 a review, or 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>]
  ],
  ...
}
```

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>]
  ],
  ...
}
```

A moderation suggestion for a chat event.

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

Assignment of a license to an event.

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

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 Notes

When using this NIP to bulk-label many targets at once, events may be deleted and a replacement may be published. We have opted not to use parameterizable/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 interoper-

ability 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-labe`
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.ontolo.categories)[https://ontolo.social/]

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 non-parameterized 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 ids + mandatory relay URL)
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)
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)
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*
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+MgTK5WjewFFumCcOZniWtOMSga9tJk1ky00tLoUUzyLnb1v
  "sig":
      "1173822c53261f8cffe7efbf43ba4a97a9198b3e402c2a1df130f42a8985a2d0d3430f4de350db184141e45ca844ab4e5364ea80f11d720e36357e1853
}
```

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":
      "a9a4e2192eede77e6c9d24ddfab95ba3ff7c03fbd07ad011fff245abea431fb4d3787c2d04aad001cb039cb8de91d83ce30e9a94f82ac3c5a2372aa129
}
```

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":
      "a9a4e2192eede77e6c9d24ddfab95ba3ff7c03fbd07ad011fff245abea431fb4d3787c2d04aad001cb039cb8de91d83ce30e9a94f82ac3c5a2372aa129
}
```

Encryption process pseudocode

```
val private_items = [
  ["p", "07caba282f76441955b695551c3c5c742e5b9202a3784780f8086fdcdc1da3a9"],
  ["a", "a55c15f5e41d5aebd236eca5e0142789c5385703f1a7485aa4b38d94fd18dcc4"],
]
val base64blob = nip04.encrypt(json.encode_to_string(private_items))
event.content = base64blob
```

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": "",
  ...
}

{
  "kind": 1984,
  "tags": [
    ["e", <eventId>, "illegal"],
    ["p", <pubkey>]
  ],
  "content": "He's insulting the king!",
  ...
}

{
  "kind": 1984,
  "tags": [
    ["p", <impersonator pubkey>, "impersonation"]
  ],
```

```
  "content": "Profile is impersonating nostr:<victim bech32 pubkey>",
  ...
}
```

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-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.

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

```
{
  "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-72

Moderated Communities (Reddit Style)

`draft` `optional`

The goal of this NIP is to create moderator-approved public communities around a topic. 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 issue an approval event `kind:4550` that links the community with the new post.

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.

```
{
  "created_at": <Unix timestamp in seconds>,
  "kind": 34550,
  "tags": [
    ["d", "<community-d-identifier>"],
    ["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>"]
  ],
  ...
}
```

New Post Request

Any Nostr event can be submitted to a community by anyone for approval. Clients MUST add the community's `a` tag to the new post event in order to be presented for the moderator's approval.

```
{
  "kind": 1,
  "tags": [
    ["a", "34550:<community event author pubkey>:<community-d-identifier>", "<optional-relay-url>"],
  ],
```

```
  "content": "hello world",
  // ...
}
```

Community management clients MAY filter all mentions to a given `kind:34550` event and request moderators to approve each submission. Moderators MAY delete his/her approval of a post at any time using event deletions (See NIP-09).

Post Approval by moderators

The post-approval event MUST include `a` tags of the communities the moderator is posting into (one or more), the `e` tag of the post and `p` tag of the author of the post (for approval notifications). The event SHOULD also include the stringified `post request` event inside the `.content` (NIP-18-style) and a `k` tag with the original post's event kind to allow filtering of approved posts by kind.

```
{
  "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>",
  // ...
}
```

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.

Post Approvals of replaceable events can be created in three ways: (i) by tagging the replaceable event as an `e` tag if moderators want to approve each individual change to the replaceable event; (ii) by tagging the replaceable event as an `a` tag if the moderator authorizes the replaceable event author to make changes without additional approvals and (iii) 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 `e`-approval 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 included in all `34550:*` `a` tags.

Displaying

Community clients SHOULD display posts that have been approved by at least 1 moderator or by the community owner.

The following filter displays the approved posts.

```
[
  "REQ",
  "_",
  {
    "authors": ["<owner-pubkey>", "<moderator1-pubkey>", "<moderator2-pubkey>", "<moderator3-pubkey>", ...],
```

```
    "kinds": [4550],
    "#a": ["34550:<Community event author pubkey>:<d-identifier of the community>"],
  }
]
```

Clients MAY hide approvals by blocked moderators at the user's request.

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 `000000000e9d97a1ab09fc38103` 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":
      "284622fc0a3f4f1303455d5175f7ba962a3300d136085b9566801bc2e0699de0c7e31e44c81fb40ad9049173742e904713c3594a1da0fc5d2382a25c11
}
```

Validating

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

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int countLeadingZeroes(const char *hex) {
    int count = 0;

    for (int i = 0; i < strlen(hex); i++) {
        int nibble = (int)strtol((char[]){hex[i], '\0'}, NULL, 16);
        if (nibble == 0) {
            count += 4;
        } else {
            count += __builtin_clz(nibble) - 28;
            break;
        }
    }

    return count;
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <hex_string>\n", argv[0]);
        return 1;
    }

    const char *hex_string = argv[1];
    int result = countLeadingZeroes(hex_string);
    printf("Leading zeroes in hex string %s: %d\n", hex_string, result);

    return 0;
}
```

Here is some JavaScript code for doing the same thing:

```javascript
// 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;
}
```

Querying relays for PoW notes

If relays allow searching on prefixes, you can use this as a way to filter notes of a certain difficulty:

```
$ echo '["REQ", "subid", {"ids": ["000000000"]}]'  | websocat wss://some-relay.com | jq -c '.[2]'
{"id":"000000000121637feeb68a06c8fa7abd25774bdedfa9b6ef648386fb3b70c387", ...}
```

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.

Relays

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>,
  "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.

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 `NIP`s 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
  },
  ...
}
```

- `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}
  ]
}
```

`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 cleartext 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 favourable zones. Exposing this flexibility is up to the client software.

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

- `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",
  ...
}
```

- `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" }],
  },
  ...
}
```

Icon

A 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",
  ...
}
```

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
      }
    ]
  },
}

\pagebreak

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:

```json
["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"]
  ],
  ...
}
```

`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-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:

```
{
  ...
  "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.

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-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-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 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.

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/9f5248`
`atproto`	AT URI	`at://did:plc:zhbjlbmir5dganqhueg7y4i3/`
`rss`	URL with guid fragment	`https://soapbox.pub/rss/feed.xml#https`
`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":
      "a72f12c08f18e85d98fb92ae89e2fe63e48b8864c5e10fbdd5335f3c9f936397a6b0a7350efe251f8168b1601d7012d4a6d0ee6eec958067cf22a14f5a
}
```

See also

- FEP-fffd: Proxy Objects
- Mostr bridge

Clients

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:nprofile1qqsrhuxx8l9ex335q7he0f09aej04zpazpl0ne2cgukyawd24mayt8gpp4mhxue69uhhytnc9e3k7mgpz4mhxue6`
- `nostr:note1fntxtkcy9pjwucqwa9mddn7v03wwwsu9j330jj350nvhpky2tuaspk6nqc`
- `nostr:nevent1qqstna2yrezu5wghjvswqqculvvwxsrcvu7uc0f78gan4xqhvz49d9spr3mhxue69uhkummnw3ez6un9d3shjtn4de`

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
- `nrelay`: a nostr relay
- `naddr`: a nostr *replaceable event* coordinate

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 `nrelay`, this is the relay URL
    * for `naddr`, it is the identifier (the `"d"` tag) of the event being referenced. For non-parameterized 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
- `nprofile1qqsrhuxx8l9ex335q7he0f09aej04zpazpl0ne2cgukyawd24mayt8gpp4mhxue69uhhytnc9e3k7mgpz4mhxue69uhkg6n` 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-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]
```

Payments

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 receipts` 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 parameterized replaceable 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":
      "f2cb581a84ed10e4dc84937bd98e27acac71ab057255f6aa8dfa561808c981fe8870f4a03c1e3666784d82a9c802d3704e174371aa13d63e2aeaf24ff5
}
```

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 his 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 receipts`.
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 receipts` 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 invoice description.
- `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",
         "lnbc10u1p3unwfusp5t9r3yymhpfqculx78u027lxspgxcr2n2987mx2j55nnfs95nxnzqpp5jmrh92pfld78spqs78v9euf2385t83uvpwk9ldrlvf6ch
     ["description",
         "{\"pubkey\":\"97c70a44366a6535c145b333f973ea86dfdc2d7a99da618c40c64705ad98e322\",\"content\":\"\",\"id\":\"d9cc14d50fc
     ["preimage", "5d006d2cf1e73c7148e7519a4c68adc81642ce0e25a432b2434c99f97344c15f"]
   ],
   "content": "",
 }
```


Appendix F: Validating Zap Receipts

A client can retrieve `zap receipts` 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-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 pay Lightning invoices.
- **user**: The person using the **client**, and want's to connect their wallet app 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 send lightning payments to other nostr users 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** makes a payment. 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.

Events

There are three event kinds: - `NIP-47 info event`: 13194 - `NIP-47 request`: 23194 - `NIP-47 response`: 23195

The info event should be a replaceable event that is published by the **wallet service** on the relay to indicate which commands it supports. The content should be a plaintext string with the supported commands, space-separated, eg. `pay_invoice get_balance`. Only the `pay_invoice` command is described in this NIP, but other commands might be defined in different NIPs.

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.

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=71a
```

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
    }
}
```

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 an `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
    }
}
```

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
    }
}
```

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": "77616c5f68724448733052424454d353736"}]},
        ],
    }
}
```

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
    }
}
```

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
    }
}
```

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.

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",
  ...
```

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",
  ...
}
```

The goal MAY include an `r` or `a` tag linking to a URL or parameterized replaceable event.

The goal MAY include multiple beneficiary pubkeys by specifying `zap` tags.

Parameterized replaceable 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)>"],
  ],
  ...
}
```

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 a parameterized replaceable 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

Third Parties

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.

```
# 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>1674834236&created_at<1677426236
```

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

```
6f44d7fe4f1c09f3954640fb58bd12bae8bb8ff4120853c4693106c82e920e2b898f1f9ba9bd65449a987c39c0423426ab7b53910c0c6abfb41b30bc16e5f524
```

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",
      "6f44d7fe4f1c09f3954640fb58bd12bae8bb8ff4120853c4693106c82e920e2b898f1f9ba9bd65449a987c39c0423426ab7b53910c0c6abfb41b30bc16e
    ]
  ],
  "content": "Hello, world!",
  "sig":
      "633db60e2e7082c13a47a6b19d663d45b2a2ebdeaf0b4c35ef83be2738030c54fc7fd56d139652937cdca875ee61b51904a1d0d0588a6acd6168d7be29
}
```

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-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+QGkPCadJTn8FxGkd24iApo3BR9/M0uw6n4RFAFSPAKKMgkzVMoRyR3ZS/aqATDFvoZJOkE9cPG/TAzmyZvr/WUIS8kLmuI1dCA+itFF6+ULZ
  "kind": 13,
  "created_at": 1703015180,
  "pubkey": "611df01bfcf85c26ae65453b772d8f1dfd25c264621c0277e1fc1518686faef9",
  "tags": [],
  "id": "28a87d7c074d94a58e9e89bb3e9e4e813e2189f285d797b1c56069d36f59eaa7",
  "sig":
      "02fc3facf6621196c32912b1ef53bac8f8bfe9db51c0e7102c073103586b0d29c3f39bdaa1e62856c20e90b6c7cc5dc34ca8bb6a528872cf6e65e62845
}
```

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+I9ViA5pxLyFfQjs1lcE6KdX3euC
  "kind": 1059,
  "created_at": 1703021488,
  "pubkey": "18b1a75918f1f2c90c23da616bce317d36e348bcf5f7ba55e75949319210c87c",
  "id": "5c005f3ccf01950aa8d131203248544fb1e41a0d698e846bd419cec3890903ac",
  "sig":
      "35fabdae4634eb630880a1896a886e40fd6ea8a60958e30b89b33a93e6235df750097b04f9e13053764251b8bc5dd7e8e0794a3426a90b6bcc7e5ff660
  "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 his/her private key.

const unwrappedSeal = nip44Decrypt(wrap, recipientPrivateKey)
const unsealedRumor = nip44Decrypt(unwrappedSeal, recipientPrivateKey)
```

NIP-46 - Nostr Remote Signing

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

- **Local keypair**: A local public and private key-pair used to encrypt content and communicate with the remote signer. Usually created by the client application.
- **Remote user pubkey**: The public key that the user wants to sign as. The remote signer has control of the private key that matches this public key.
- **Remote signer pubkey**: This is the public key of the remote signer itself. This is needed in both `create_account` command because you don't yet have a remote user pubkey.

All pubkeys specified in this NIP are in hex format.

Initiating a connection

To initiate a connection between a client and a remote signer there are a few different options.

Direct connection initiated by remote signer

This is most common in a situation where you have your own nsecbunker or other type of remote signer and want to connect through a client that supports remote signing.

The remote signer would provide a connection token in the form:

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

This token is pasted into the client by the user and the client then uses the details to connect to the remote signer via the specified relay(s).

Direct connection initiated by the client

In this case, basically the opposite direction of the first case, the client provides a connection token (or encodes the token in a QR code) and the signer initiates a connection to the client via the specified relay(s).

```
nostrconnect://<local-keypair-pubkey>?relay=<wss://relay-to-connect-on>&metadata=<json metadata in the form:
    {"name":"...", "url": "...", "description": "..."}>
```

The flow

1. Client creates a local keypair. This keypair doesn't need to be communicated to the user since it's largely disposable (i.e. the user doesn't need to see this pubkey). Clients might choose to store it locally and they should delete it when the user logs out.
2. Client gets the remote user pubkey (either via a `bunker://` connection string or a NIP-05 login-flow; shown below)
3. Clients use the local keypair to send requests to the remote signer by `p`-tagging and encrypting to the remote user pubkey.
4. The remote signer responds to the client by `p`-tagging and encrypting to the local keypair pubkey.

Example flow for signing an event

- Remote user pubkey (e.g. signing as) `fa984bd7dbb282f07e16e7ae87b26a2a7b9b90b7246a44771f0cf5ae58018f52`
- Local pubkey is `eff37350d839ce3707332348af4549a96051bd695d3223af4aabce4993531d86`

```
{
    "kind": 24133,
    "pubkey": "eff37350d839ce3707332348af4549a96051bd695d3223af4aabce4993531d86",
    "content": nip04({
        "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 user
        pubkey
}
```

```
{
    "kind": 24133,
    "pubkey": "fa984bd7dbb282f07e16e7ae87b26a2a7b9b90b7246a44771f0cf5ae58018f52",
    "content": nip04({
        "id": <random_string>,
        "result": json_stringified(<signed-event>)
    }),
    "tags": [["p", "eff37350d839ce3707332348af4549a96051bd695d3223af4aabce4993531d86"]], // p-tags the local keypair
        pubkey
}
```

Diagram

Request Events `kind: 24133`

```
{
    "id": <id>,
    "kind": 24133,
    "pubkey": <local_keypair_pubkey>,
    "content": <nip04(<request>)>,
    "tags": [["p", <remote_user_pubkey>]], // NB: in the `create_account` event, the remote signer pubkey should be
        `p` tagged.
    "created_at": <unix timestamp in seconds>
}
```

The `content` field is a JSON-RPC-like message that is NIP-04 encrypted and has the following structure:

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

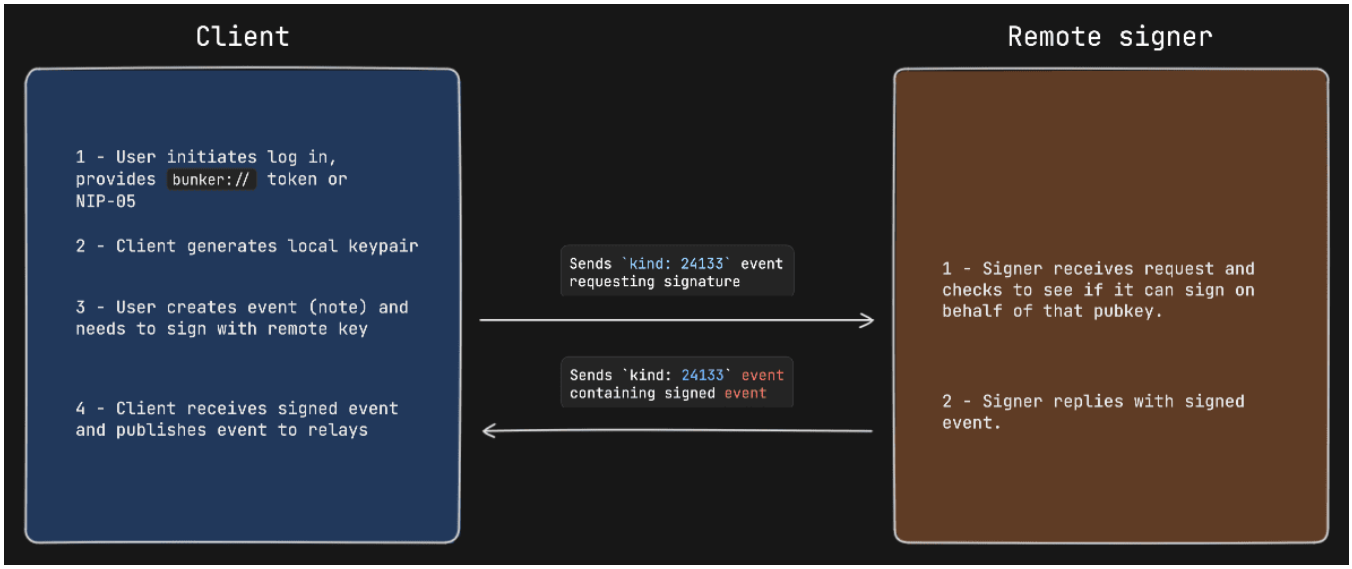

Figure 1: signing-example

- `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_user_pubkey>, <optional_secret>, <optional_requested_permissions>]`	"ack"
`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`	`[]`	`<hex-pubkey>`
`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. `nip04_encrypt,sign_event:4` meaning permissions to call `nip04_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.

Response Events `kind:24133`

```
{
    "id": <id>,
    "kind": 24133,
    "pubkey": <remote_signer_pubkey>,
    "content": <nip04(<response>)>,
    "tags": [["p", <local_keypair_pubkey>]],
    "created_at": <unix timestamp in seconds>
}
```

The `content` field is a JSON-RPC-like message that is NIP-04 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.

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. This is currently used in the OAuth-like flow enabled by signers like Nsecbunker. The response `content` object will take the following form:

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

Clients 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). It's also possible to add a `redirect_uri` url parameter to the auth_url, which is helpful in situations when a client cannot open a new window or tab to display the auth challenge.

Example event signing request with auth challenge

Remote Signer Commands

Remote signers might support additional commands when communicating directly with it. These commands follow the same flow as noted above, the only difference is that when the client sends a request event, the p-tag is the pubkey of the remote signer itself and the `content` payload is encrypted to the same remote signer pubkey.

Methods/Commands

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

Command	Params	Result
`create_account`	`[<username>, <domain>, <optional_email>, <optional_requested_permissions>]`	`<newly_created_remote_user_pubkey>`

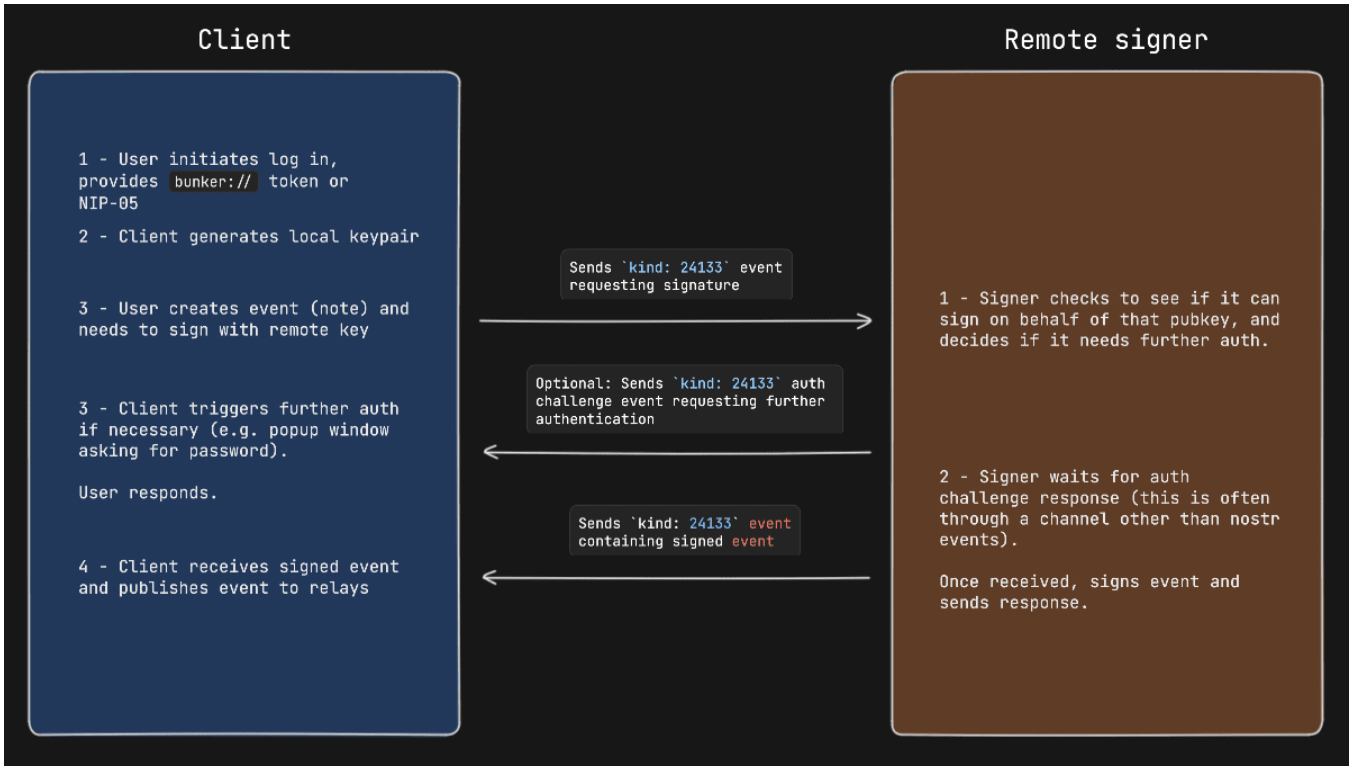

Figure 2: signing-example-with-auth-challenge

Appendix

NIP-05 Login Flow

Clients might choose to present a more familiar login flow, so users can type a NIP-05 address instead of a `bunker://` string.

When the user types a NIP-05 the client:

- Queries the `/.well-known/nostr.json` file from the domain for the NIP-05 address provided to get the user's pubkey (this is the **remote user pubkey**)
- In the same `/.well-known/nostr.json` file, queries for the `nip46` key to get the relays that the remote signer will be listening on.
- Now the client has enough information to send commands to the remote signer on behalf of the user.

OAuth-like Flow

Remote signer discovery via NIP-89 In this last case, most often used to facilitate an OAuth-like signin flow, the client first looks for remote signers that have announced themselves via NIP-89 application handler events.

First the client will query for `kind: 31990` events that have a `k` tag of `24133`.

These are generally shown to a user, and once the user selects which remote signer to use and provides the remote user pubkey they want to use (via npub, pubkey, or nip-05 value), the client can initiate a connection. Note that it's on the user to select the remote signer that is actually managing the remote key that they would like to use in this case. If the remote user pubkey is managed on another remote signer, the connection will fail.

In addition, it's important that clients validate that the pubkey of the announced remote signer matches the pubkey of the _ entry in the `/.well-known/nostr.json` file of the remote signer's announced domain.

Clients that allow users to create new accounts should also consider validating the availability of a given username in the namespace of remote signer's domain by checking the `/.well-known/nostr.json` file for existing usernames. Clients can then show users feedback in the UI before sending a `create_account` event to the remote signer and

receiving an error in return. Ideally, remote signers would also respond with understandable error messages if a client tries to create an account with an existing username.

Example Oauth-like flow to create a new user account with Nsecbunker Coming soon…

References

- NIP-04 - Encryption

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" ]
    ]
}
```

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"]
  ],
  ...
}
```

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>"]
  ],
  ...
}
```

- `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"]
  ],
  ...
}
```

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>"],
  ],
  ...
}
```

- `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"
  ],
  ...
}
```

Customers can use NIP-89 to see what service providers their follows use.

Application Features

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 *parameterized replaceable* 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 a parameterized replaceable event 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 a parameterized replaceable 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. * `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": 31923,
  "content": "<description of calendar event>",
  "tags": [
    ["d", "<UUID>"],

    ["title", "<title of calendar event>"],

    // 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>"],

    // 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.

Format

The format uses a parameterized replaceable 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) reference tag to 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`.

```
{
  "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": [
    ["a", "<31922 or 31923>:<calendar event author pubkey>:<d-identifier of calendar event>", "<optional relay
        url>"],
    ["d", "<UUID>"],
    ["status", "<accepted/declined/tentative>"],
    ["fb", "<free/busy>"],
  ]
}
```

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 a *parameterized replaceable 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": "",
  ...
}
```

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.",
  ...
}
```

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":
      "5bc7a60f5688effa5287244a24768cbe0dcd854436090abc3bef172f7f5db1410af4277508dbafc4f70a754a891c90ce3b966a7bc47e7c1eb71ff57640
}
```

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":
      "997f62ddfc0827c121043074d50cfce7a528e978c575722748629a4137c45b75bdbc84170bedc723ef0a5a4c3daebf1fef2e93f5e2ddb98e5d685d022c
}
```

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"]
  ],
  ...
}
```

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-15

Nostr Marketplace

`draft` `optional`

Based on https://github.com/lnbits/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]],
  ...
}
```

- 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],
    ...
  ],
  ...
```

- 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)],
  ...
}
```

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>]],
}
```

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 "parameterized replaceable 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 https://github.com/nostr-protocol/nips/blob/master/04.md.

Additional

Standard data models can be found here

NIP-99

Classified Listings

`draft` `optional`

This NIP defines `kind:30402`: a parameterized replaceable 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, as per NIP-12.
- 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:nevent1qqst8cujky046negxgwwm5ynqwn53t8aqjr6afd8g59nfqwxpdhylpcpzamhxue69uhhyetvv9ujuetcv9khqmr99e3k7mg8arnc9]
      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:naddr1qqzkjurnw4ksz9thwden5te0wfjkccte9ehx7um5wghx7un8qgs2d90kkcq3nk2jry62dyf50k0h36rhpdtd594my40w9pkal876jxgrqsqqqa2
  "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": "..."
}
```

NIP-54

Wiki

`draft` `optional`

This NIP defines `kind:30818` (a *parameterized replaceable event*) for long-form text content similar to NIP-23, but with one important difference: articles are meant to be 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 rules

The content should be Markdown, following the same rules as of NIP-23, although it takes some extra (optional) metadata 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.

One extra functionality is added: **wikilinks**. Unlike normal Markdown links `[]()` that link to 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`.

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 Markdown?

If the idea is to make a wiki then the most obvious text format to use is probably the mediawiki/wikitext format used by Wikipedia since it's widely deployed in all mediawiki installations and used for decades with great success. However, it turns out that format is very bloated and convoluted, has way too many features and probably because of that it doesn't have many alternative implementations out there, and the ones that exist are not complete and don't look very trustworthy. Also it is very much a centralized format that can probably be changed at the whims of the Wikipedia owners.

On the other hand, Markdown has proven to work well for small scale wikis and one of the biggest wikis in the planet (which is not very often thought of as a wiki), StackOverflow and its child sites, and also one of the biggest "personal wiki" software, Obsidian. Markdown can probably deliver 95% of the functionality of wikitext. When

augmented with tables, diagram generators and MathJax (which are common extensions that exist in the wild and can be included in this NIP) that rate probably goes to 99%, and its simplicity is a huge benefit that can't be overlooked. Wikitext format can also be transpíled into Markdown using Pandoc. Given all that, I think it's a reasonable suspicion that mediawiki is not inherently better than Markdown, the success of Wikipedia probably cannot be predicated on the syntax language choice.

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 (the pubkey being requested to merge something into their article can create [NIP-25] reactions that tag the `kind:818` event with + or -

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.

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"], // ommited 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.

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

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"],
    // ...
  ]
}
```

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-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 type, 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

```
{
  "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>],
    ["image", <string with preview URI>],
    ["summary", <excerpt>],
    ["alt", <description>]
  ],
  "content": "<caption>",
  ...
}
```

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.

Upload

A file can be uploaded one at a time to `https://your-file-server.example/custom-api-path` (route from `https://your-file-server.example/.well-known/nostr/nip96.json` "api_url" field) as `multipart/form-data` content type using `POST` method with the file object set to the `file` form data field.

`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). If using an html form, use an `Authorization` form data field instead.

These following **optional** form data fields MAY be used by `servers` and SHOULD be sent by `clients`: - `expiration`: string of the UNIX timestamp in seconds. Empty string if file should be stored forever. The server isn't required to honor this; - `size`: string of the 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; - `caption`: loose description; - `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.

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`.

Note for `clients`: if using an HTML form, it is important for the `file` form field to be the **last** one, or be re-ordered right before sending or be appended as the last field of XHR2's FormData object.

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 (which is embedded in the decoded header value) as the owner of the file so to later allow them to delete the file. Note that if a file with the same hash of a previously received file (so the same file) is uploaded by another user, the server doesn't need to store the new file. It should just add the new user's `pubkey` to the list of the owners of the already stored file with said hash (if it wants to save space by keeping just one copy of the same file, because multiple uploads of the same file results in the same file hash).

The `server` MAY also store the `Authorization` header/field value (decoded or not) for accountability purpose as this proves that the user with the unique pubkey did ask for the upload of the file with a specific hash. However, storing the pubkey is sufficient to establish ownership.

The `server` MUST reject with 413 Payload Too Large if file size exceeds limits.

The `server` MUST reject with 400 Bad Request status if some fields are invalid.

The `server` MUST reply to the upload with 200 OK status if the `payload` tag value contains an already used SHA-256 hash (if file is already owned by the same pubkey) or reject the upload with 403 Forbidden status if it isn't the same of the received file.

The `server` MAY reject the upload with 402 Payment Required status if the user has a pending payment (Payment flow is not strictly required. Server owners decide if the storage is free or not. Monetization schemes may be added later to correlated NIPs.).

On successful uploads the `server` MUST reply with **201 Created** HTTP status code or **202 Accepted** if a `processing_url` field is added to the response so that the `client` can follow the processing status (see Delayed Processing section).

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.

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

`Servers` must make available the route `https://your-file-server.example/custom-api-path/<sha256-file-hash>(.ext)` (route taken from `https://your-file-server.example/.well-known/nostr/nip96.json` "api_url" or "download_url" field) with `GET` method for file download.

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-file-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: `https://your-file-server.example/custom-api-path/719171db19525d9d08dd69cb716a18158a249b7b3b3ec4bbdec`

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: `https://your-file-server.example/custom-api-path/<sha256-file-hash>.png?w=32`

Deletion

`Servers` must make available the route `https://deletion.domain/deletion-path/<sha256-file-hash>(.ext)` (route taken from `https://your-file-server.example/.well-known/nostr/nip96.json` "api_url" field) with `DELETE` method for file deletion.

Note that the "<sha256-file-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.

`Clients` should send a `DELETE` request to the server deletion route in the above format. It must include a NIP-98 `Authorization` header.

The `server` should reject deletes from users other than the original uploader. The `pubkey` encoded on the header value identifies the user.

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."
}
```

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"]
  ]
}
```

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` (parameterized replaceable 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

Security

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-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: "ÅΩḭ" - 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: "ÅΩḭ" - 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:

`ncryptsec1qgg9947rlpvqu76pj5ecreduf9jxhselq2nae2kghhvd5g7dgjtcxfqtd67p9m0w57lspw8gsq6yphnm8623nsl8xn9j4jdzz8`

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-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":
      "5ed9d8ec958bc854f997bdc24ac337d005af372324747efe4a00e24f4c30437ff4dd8308684bed467d9d6be3e5a517bb43b1732cc7d33949a3aaf86705
}
```

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
    eyJpZCI6ImZlOTY0ZTc1ODkwMzM2MGYyOGQ4NDI0ZDA5MmRhODQ5NGVkMjA3Y2JhODIzMTEwYmUzYTU3ZGZlNGI1Nzg3MzQiLCJwdWJrZXki0iI2M2ZlNjMxOGRjN
```

Reference Implementations

- C# ASP.NET `AuthenticationHandler` NostrAuth.cs

Developers

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
```

Implementation

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

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-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"]
  ]
}
```

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>"]
  ]
}
```

- `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"]
  ]
  ...
}
```

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"]
  ],
  ...
}
```

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': [...] }]']
```

Conclusion

Thank you for exploring the nostr-book. My hope is that this reorganized collection of Nostr Notes in Progress (NIPs) has provided you with a clearer and more structured understanding of the Nostr protocol. By grouping similar NIPs together, the aim was to create a logical flow that enhances comprehension and makes the information more accessible to everyone.

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

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

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