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!

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. 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.

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` can be anything and should be ignored.

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-03

OpenTimestamps Attestations for Events

`draft` `optional`

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

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

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

Example OpenTimestamps proof verification flow

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

```
~> nak req -i e71c6ea722987debdb60f81f9ea4f604b5ac0664120dd64fb9d23abc4ec7c323 wss://nostr-pub.wellorder.net | jq -r
     .content | ots verify
> using an esplora server at https://blockstream.info/api
- sequence ending on block 810391 is valid
timestamp validated at block [810391]
```

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

NIP-04

Encrypted Direct Message

`final` `unrecommended` `optional`

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

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

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

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

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

Code sample for generating such an event in JavaScript:

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

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

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

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

Security Warning

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

Client Implementation Warning

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

NIP-05

Mapping Nostr keys to DNS-based internet identifiers

`final` `optional`

On events of kind `0` (`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-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-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>]`

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"`.

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>`.

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-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-13
======

Proof of Work

`draft` `optional`

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

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

```
    bits.

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

Mining
------

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

```json
{"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", "21"]
  ],
  "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:

```
// 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.

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-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.

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-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 efficency
    ["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-42

Authentication of clients to relays

`draft` `optional`

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

Motivation

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

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

Definitions

New client-relay protocol messages

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

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

And, when sent by clients, the following form:

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

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

Canonical authentication event

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

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

`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-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]
}
```

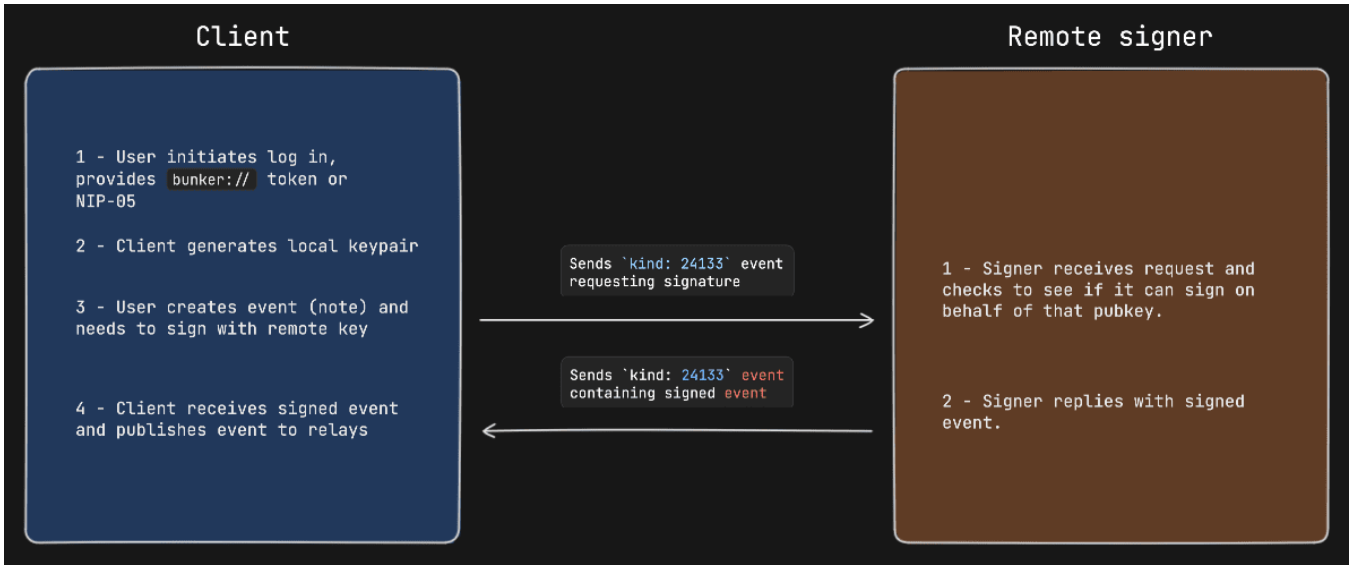

Figure 1: signing-example

- `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>`

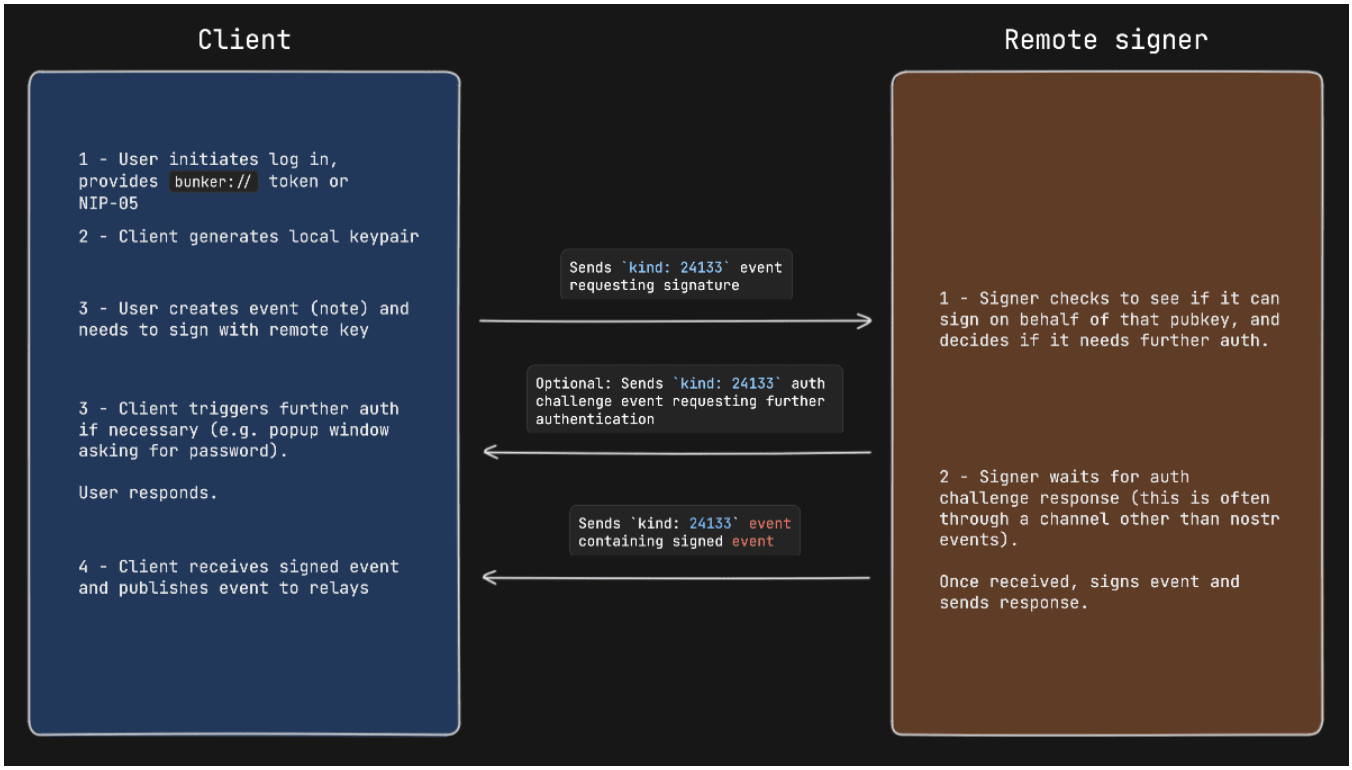

Figure 2: signing-example-with-auth-challenge

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 fascilitate 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-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-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 live.snort.social and nostrnests.com.

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-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.

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 indepedent 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-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.

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!