Content addressing and CIDs

> For a deep dive into how Content Identifiers (CIDs) are constructed, take a look at ProtoSchool's tutorial on the Anatomy of a CID .

- Identifier formats
  - Version 0 (v0)
  - Version 1 (v1)
- CID Inspector
- CID conversion
  - v0 to v1
  - Text to binary
  - CID to hex
- Further resources

A *content identifier*, or CID, is a label used to point to material in IPFS. It doesn't indicate *where* the content is stored, but it forms a kind of address based on the content itself. CIDs are short, regardless of the size of their underlying content.

CIDs are based on the content's cryptographic hash. That means:

- Any difference in the content will produce a different CID and
- The same content added to two different IPFS nodes using the same settings will produce *the same CID*.

IPFS uses the `sha-256` hashing algorithm by default, but there is support for many other algorithms. The Multihash project represents the work for this, with the aim of future-proofing applications' use of hashes and allowing multiple hash functions to coexist. (If you're curious about how hash types in IPFS are decided upon, you may wish to keep an eye on this forum discussion .)

Identifier formats

CIDs can take a few different forms with different encoding bases or CID versions. Many of the existing IPFS tools still generate v0 CIDs, although the `files` (**Mutable File System**) and `object` operations now use CIDv1 by default.

Version 0 (v0)

When IPFS was first designed, we used base 58-encoded multihashes as the content identifiers. This is simpler but much less flexible than newer CIDs. CIDv0 is still used by default for many IPFS operations, so you should generally support v0.

If a CID is 46 characters starting with "Qm", it's a CIDv0 (for more details, check the **decoding algorithm** in the CID specification).

Version 1 (v1)

CID v1 contains some leading identifiers that clarify exactly which representation is used, along with the content-hash itself. These include:

- A **multibase** prefix, specifying the encoding used for the remainder of the CID
- A CID version identifier, which indicates which version of CID this is
- A **multicodec** identifier, indicating the format of the target content — it helps people and software to know how to interpret that content after the content is fetched

These leading identifiers also provide forward-compatibility, supporting different formats to be used in future versions of CID.

You can use the first few bytes of the CID to interpret the remainder of the content address and know how to decode the content after being fetched from IPFS. For more details, check out the **CID specification** . It includes a **decoding algorithm** and links to existing software implementations for decoding CIDs.

If you can't decide between CIDv0 and CIDv1, consider choosing CIDv1 for your new project and opt in by passing a version flag (`ipfs add --cid-version 1`). This is more future-proof and **safe for use in browser contexts**.

The IPFS project will switch to CIDv1 as the new default in the near future.

CID Inspector

It's easy to explore a CID for yourself. Want to pull apart a specific CID's multibase, multicodec, or multihash info? You can use the CID Inspector or the CID Info panel in IPLD Explorer (both links launch using a sample CID) for an interactive breakdown of differently-formatted CIDs.

Check out ProtoSchool's Anatomy of a CID tutorial to see how a single file can be represented in multiple CID versions.

CID conversion

Converting a CID from v0 to v1 enables it to be represented in multibase encodings.
The default for CIDv1 is the case-insensitive `base32` , but use of the shorter `base36` is encouraged for IPNS names to ensure same text representation on subdomains.

v0 to v1

The built-in `ipfs cid format` command can be used from the command line:

```
$ ipfs cid format -v 1 -b base32 QmbWqxBEKC3P8tqsKc98xmWNzrzDtRLMiMPL8wBuTGsMnR
bafybeigdyrzt5sfp7udm7hu76uh7y26nf3efuylqabf3oclgtqy55fbzdi
```

JavaScript users can also leverage the `toV1()` method provided by the `cids` library:

```js
const CID = require('cids')
new CID('QmbWqxBEKC3P8tqsKc98xmWNzrzDtRLMiMPL8wBuTGsMnR').toV1().toString('base32')
// → bafybeigdyrzt5sfp7udm7hu76uh7y26nf3efuylqabf3oclgtqy55fbzdi
```

Text to binary

A CID can be represented as both text and as a stream of bytes. The latter may be a better choice when speed and storage efficiency are considerations.

To convert a CIDv1 from text to binary form, simply read the first character
and then decode the remainder using the encoding specified in the multibase table .

JS users can leverage the `cids` library to get a binary version as `Uint8Array` :

```js
const CID = require('cids')
new CID('bafybeigdyrzt5sfp7udm7hu76uh7y26nf3efuylqabf3oclgtqy55fbzdi').bytes
```

```
// → Uint8Array [ 1, 112,  18,  32, 195, 196, 115,  62, ... ]
```

Be mindful about parsing CIDs correctly. Avoid shortcuts.

Unless you are the one who imported the data to IPFS, the length of a CID is not deterministic and depends on the length of the multihash inside of it.

To illustrate, passing a custom hash function will produce CIDs of varying lengths:

```
$ ipfs add --cid-version 1 --hash sha2-256    -nq cat.jpg | wc -c
60
$ ipfs add --cid-version 1 --hash blake2b-256 -nq cat.jpg | wc -c
63
$ ipfs add --cid-version 1 --hash sha3-512    -nq cat.jpg | wc -c
111
```

CID to hex

Sometimes, a hexadecimal representation of raw bytes is prefered for debug purposes. To get the hex for raw `.bytes` of an entire CID, one can use built-in support for `base16` encoding and skip the `f` (multibase prefix):

```js
> cid.toString('base16')
'f01701220c3c4733ec8affd06cf9e9ff50ffc6bcd2ec85a6170004bb709669c31de94391a'

> cid.toString('base16').substring(1)
'01701220c3c4733ec8affd06cf9e9ff50ffc6bcd2ec85a6170004bb709669c31de94391a' // "cid as hex"
```

To convert back to a CIDv1, prepend the hex value with `f` (multibase prefix for lowercase base16). Use it as-is (it is a valid CID), or convert to a different multibase by passing it as an argument to `toString`:

```js
> new CID('f' +'01701220c3c4733ec8affd06cf9e9ff50ffc6bcd2ec85a6170004bb709669c31de94391a').toS
// → bafybeigdyrzt5sfp7udm7hu76uh7y26nf3efuylqabf3oclgtqy55fbzdi
```

TIP

> Subdomain gateways convert paths with custom bases like base16 to base32 or base36, in an effort to fit a CID in a DNS label:
>
> - dweb.link/ipfs/f01701220c3c4733ec8affd06cf9e9ff50ffc6bcd2ec85a6170004bb709669c31de94391a
>   returns a HTTP 301 redirect:
>   → bafybeigdyrzt5sfp7udm7hu76uh7y26nf3efuylqabf3oclgtqy55fbzdi.ipfs.dweb.link

Further resources

Check out these links for more information on CIDs and how they work:

- Core Course: How IPFS Deals With Files
- Files and IPFS Companion

Was this information helpful?

Yes No

Edit this page on GitHub or open an issue

Help us improve this site!

Suggest new content

Request features

Give general feedback