# **What is IPFS?**

IPLD is the data model of the content-addressable web. It allows us to treat all hash-linked data structures as subsets of a unified information space, unifying all data models that link data with hashes as instances of IPLD.

# **WHY IPLD?**

## A data model for interoperable protocols.

Content addressing through hashes has become a widely-used means of connecting data in distributed systems, from the blockchains that run your favorite cryptocurrencies, to the commits that back our code, to the web’s content at large. Yet, whilst all of these tools rely on some common primitives, their specific underlying data structures are not interoperable.

IPLD is a single namespace for all hash-inspired protocols. Through IPLD, links can be traversed across protocols, allowing you explore data regardless of the underlying protocol.



# **Use Case**

## Ethereum to IPFS

We can have our contract refer to media on IPFS, perhaps modifying it and tracking its changes on each function execution. Seamlessly jump from function to object through IPLD addressing.



# **Features**

## A Canonical Data Model

A self-contained descriptive model that uniquely identifies any hash-based data structure and ensures the same logical object always maps to the exact same sequence of bits.

## Protocol independent resolution

IPLD brings isolated systems together, making integration with existing protocols simple.

## Upgradeable

With Multiformats support, IPLD is easily upgradeable and will grow with your favorite protocols.

## Operates across formats

Express your IPLD objects in various serializable formats like JSON, CBOR, YAML, XML and many more, making IPLD a cinch to use with any framework.

## Backwards compatible

Non-intrusive resolvers make IPLD easy to integrate within your existing work.

## A namespace for all protocols

IPLD allows you to explore data across protocols seamlessly, binding hash-based data structures together through a common namespace.

# **Working with IPLD using CLI Commands**

In order to make this work **we must have** **IPFS** installed in our system.

1. **ipfs dag put:**

It accepts input from a file or stdin and parses it into an object of the specified format. It returns the **Content ID(CID)** of the said file or stdin.



1. **Ipfs dag get:**

It takes the CID and fetches a dag node from ipfs and prints it out in the specified format.



We can also get the value of a specific key by providing the key name along with the CID as given below,



1. **Ipfs dag resolve:**

It fetches a dag node from ipfs, prints its address and remaining path.

# **Working with IPLD using HTTP API**

We must have **Node.js** installed before proceeding. Then install the following dependencies: -

* **npm install ipfs-http-client cids --save**

We can create a new node by passing a data object into the ipfs.dag.put method, which returns a Content Identifier (CID) for the newly created node.

* **ipfs.dag.put ({name: ‘john’})**

A **CID is an address** for a block of data in IPFS that is derived from its content. Every time someone puts the same {name: john’} data into IPFS, they'll get back an identical CID to the one you got. If they put in {name: ‘jOhn’} instead, the CID will be different. The CID will look like ***zdpuAxmnJywkZ1UG5LQuUTxDDkMjUyPi1fMpE8GPyXh3ozCHg.***

* **ipfs.dag.get (CID)** where CID iszdpuAxmnJywkZ1UG5LQuUTxDDkMjUyPi1fMpE8GPyXh3ozCHg

It allows queries using IPFS paths. These queries return an object containing the value of the query and any remaining path that was unresolved.

# **Echoupal Example**

Below are the Javascript functions used to add, update and fetch IPLD objects using the APIs mentioned above in case of the Echoupal Project.

const ipfsClient = require('ipfs-http-client');

const CID = require('cids');

const ipfs = new ipfsClient ();

addNewObject: async function (farmerID, fieldID, classificationID, hash) {

var farmingData = await ipfs.dag.put (

{

[farmerID]:

[{

[fieldID]:

{

[classificationID]: [hash]

}

}]

}

);

const multihash = farmingData.multihash;

const cids = new CID (1, 'dag-cbor', multihash);

return cids.toBaseEncodedString();

},

updateExistingObject: async function (obj) {

var farmingData = await ipfs.dag.put(obj);

const multihash = farmingData.multihash;

const cids = new CID (1, 'dag-cbor', multihash);

return cids.toBaseEncodedString();

},

fetchObject: async function (CID) {

var record = await ipfs.dag.get(CID);

return record.value;

}

The given functions can be modified as per the requirements pertaining to the use case.