Store Data

Learn how to store data on the Filecoin network using different mechanisms that suit your project's requirements.

Prepare data for Filecoin storage

A CAR file is a standardized format for bundling and exchanging content-addressable data. It provides a way to organize and encapsulate data, ensuring it can be easily verified and retrieved.

Before sending data to Filecoin storage providers, it is necessary to package the data into CAR (Content Addressable aRchive) files, regardless of whether you store the data via a smart contract or data onramp toolings.

To provide the data to the SP which we make storage deals with, we need to prepare data and provide the following information when making storage deals via smart contracts or aggregators.

- Piece CID & Payload CID
- CAR size & piece size
- URL to your file
-

Ingredients

We can use the following tools to prepare your data into CAR for storage via FVM.

- FVM DNOt ata Depot
-
    - powered bylighthouse.storage
- CAR librariesweb3.storage/ipfs-car
- oripld/car
- IPFS node
- Store data on the IPFS network and provide CID to Filecoin SPs to initialize storage deals.
-

Instructions

We will explain each option available for preparing your data into CAR files and obtaining the necessary information to initialize storage deals via FVM, as there are multiple ways to accomplish this.

1. FVM Data Depot
2. -recommended
3.

Upload files, generate CAR, and get CAR links - we can do all these on the FVM Data Depot website. After logging in and uploading files following thistutorial , we will get the following information for proposing a storage deal via smart contract.

- Piece CID & Payload CID
- CAR size & piece size
- URL to your file
-

- using

- web3.storage/ipfs-car
- library
-

ipfs-car is a thin wrapper over@ipld/car andunix-fs which provides a library and CLI tool to pack and unpack CAR(Content Addressable aRchives) files.

After installingipfs-car via NPM, we can use it as a CLI or JS library to pack your data into a CAR file. You can refer toipfs-car GitHub to learn more about how to use it.

Pack files using CLI

Replace the file path and output path of the file you want to pack into CAR.

```

Copy

# ipfs-car will wrap files in an IPFS directory by default. --no-wrap will avoid it.

ipfs-car--pack path/to/file--no-wrap--output path/to/write/a.car

```

Expect output same as following:

```

Copy root CID: bafybeigj6nccb4rc6cujxwojt4yd7ikxxs2yekjo4zhb25ql65jh3k35um output: a.car

```

Then we can uploada.car file to the ipfs usinglighthouse.storage or IPFS desktop, and then provide the CID & URL for proposing storage deals via FVM on the Filecoin network.

1. upload to IPFS Desktop
2.

Another option is to upload data to the IPFS network using an IPFS node, such as IPFS Desktop or Kubo. By following thistutorial , you can learn how to add files using IPFS Desktop.

Afterward, you can obtain the CID or URL of the uploaded data to propose storage deals via FVM on the Filecoin network.

Store large data with the smart contract

With the support of FVM, applications can leverage the decentralized nature of the smart contract to store data on Filecoin in a more decentralized way. By initiating storage deals through smart contracts on the FVM, the Client Contract (CC) FRC is utilized to propose deals to the Filecoin network. Service Providers (SPs) running Boost can actively monitor and process these deal proposals by listening for specific smart contract events.

Client Contract serves as a crucial component in making on-chain storage deal proposals on the Filecoin network. To initialize a storage deal proposal via the Client Contract smart contract, we need to first pack your data into CAR files and obtain the following information before calling the CC smart contract.

- pieceCID
- CarLink
- car size
- piece Size
- starting and ending epoch
-

Ingredients

- Client Contract
-
    - FRC-0068
-
    - Reference Implementation
- *

-

Instructions

The Client Contract library implements the basic functions to make storage deal proposals as well as callback functions for successful storage deal creation.

One of the key methods within this library is themakeDealProposal method, which is responsible for initiating a fully on-chain storage deal proposal. To invoke themakeDealProposal method, you will need to interact with the deployed DealClient contract on Calibration. This method accepts the required parameters for the storage deal, such as the data CID or URL, car size, piece size, the duration of the deal, and any other relevant details specific to your use case.

Additionally, the Client Contract library provides callback functions that can be used to handle successful storage deal creation events. These callbacks allow you to perform actions or trigger subsequent processes upon the successful establishment of a storage deal.

A Javascript function to invoke themakeDealProposal method should be like:

```

Copy importcontractfrom"../contracts/DealClient.json"; // ... other code consthandleSubmit=async()=>{ constcontractAddress='0x0219eB1740C315fe5e20612D7E13AE2A883dB3f4';// Deployed DealClient Contract address constcontractABI=contract.abi;// the path where the DealClient.json is constcommP='baga6ea4seaqpi75umesad5vlyzyf66vbzntoave4bebmkcqu4f6nq6rchhx3ckq'; // This handles proposing storage deals try{ const{ethereum}=window; if(ethereum) { constprovider=newethers.BrowserProvider(ethereum); constsigner=awaitprovider.getSigner(); dealClient=newethers.Contract( contractAddress, contractABI, signer );

letcid=newCID(commP); constextraParamsV1=[ "https://bafybeif74tokne4wvxsrcsxh6dhrzv6ys7mtifhwzaen7jfjuvltean32a.ipfs.w3s.link/ipfs/bafybeif74tokne4wvxsrcsxh6dhrzv6ys7mtifhwzaen7jfjuvltean32a/baga6ea4seaqesm5ghdwocotmdavlrrzssfl33xh 236445,// carSize false,// skipIpniAnnounce, false,// removeUnsealedCopy ];

constDealRequestStruct=[ cid.bytes,//cidHex 8388608,//pieceSize, false,//verifiedDeal, commP,//label, 520000,// startEpoch 1555200,// endEpoch 0,// storagePricePerEpoch, 0,// providerCollateral, 0,// clientCollateral, 1,// extraParamsVersion, extraParamsV1, ]; consttransaction=awaitdealClient.makeDealProposal(DealRequestStruct); constreceipt=awaittransaction.wait(); console.log(receipt);

dealClient.on("DealProposalCreate",(id,size,verified,price)=>{ console.log(id,size,verified,price); })

console.log("Deal proposed! CID: "+cid); }else{ console.log("Ethereum object doesn't exist!"); } }catch(error) { console.log(error); return; } };

```

The full tutorial of proposal storage deals through the client contract can be foundhere .

Store small data with storage onramps

Filecoin is primarily designed for storing large data over extended periods. Due to economic considerations, it is generally not good for Service Providers (SPs) to accept small-scale datasets and allocate them to their 32 or 64 Gib storage sectors. As a result, it is unlikely that SPs will directly accept storage deals proposed by the client contract for small datasets.

In the case of small datasets, a more viable option is to store them withstorage onramps . Storage onramps combine multiple small datasets into a larger dataset and generate Proof of Deal Sub-piece Inclusion (PoDSI). PoDSI can be utilized to verify and provide evidence that the sub-piece datasets are included in a storage deal on the Filecoin network.

One of the storage onramps we can use isLighthouse.storage which is a perpetual file storage protocol that provides both on-chain and off-chain deal aggregation services. It provides a solution for storing small datasets on Filecoin while also enabling verification of deal inclusion using PoDSI. This combination of services can be valuable for ensuring the integrity and accessibility of small datasets stored on the Filecoin network.

Ingredients

- Lighthouse.storage
-
  - SDK
-
  - : a JavaScript library that allows you to upload files to the Filecoin network.
-
  - smart contract
-
  - : solidity contract to submit and process storage deal aggregation requests.
- *
-

Instructions

Lighthouse.storage provides users with two options for uploading data and making storage: utilizing the Lighthouse SDK to store data or leveraging smart contracts to initiate on-chain storage deals.

1. store data with lighthouse SDK
2.

By creating an account with Lighthouse storage and generating an API key, you can easily upload data to the Filecoin network using the Lighthouse SDK within any JavaScript application. Data stored using lighthouse SDK will be automatically registered for deal aggregation as well as RaaS(replication, renewal, and repair).

First, install lighthouse SDK in your project with the commandnpm install -g @lighthouse-web3/sdk . Then use the following code to upload data to the lighthouse for deal aggregation.

```

Copy importlighthousefrom"@lighthouse-web3/sdk"; // ... other code constfilePath='/path/to/your/files';// change the path of your file constAPIKey='YOUR_API_KEY';// the API key from the lighthouse account constuploadResponse=awaitlighthouse.upload(filePath,APIKey);

```

The expected output ofuploadResponse .

```

Copy { data:{ Name:'a.jpg', Hash:'QmUHDKv3NNL1mrg4NTW4WwJqetzwZbGNitdjr2G6Z5Xe6s', Size:'31735' } }

```

1. store data via lighthouse smart contract
2.

Lighthouse has also implemented an aggregator smart contract based onIAggregatorOracle . This smart contract is deployed on the Filecoin Calibration testnet, allowing users to submit deal aggregation requests on-chain.

We can call the smart contract at0x01ccBC72B2f0Ac91B79Ff7D2280d79e25f745960 and submit a CID for aggregation viasubmit(bytes memory _cid) external returns (uint256) methods.

A Javascript function to invoke thesubmit method should be like:

```

Copy importcontractfrom"../contracts/DealStatus.json"; importCIDfrom"cids"; // ... other code constSubmitDealAggregation=async()=>{ constcontractAddress='0x01ccBC72B2f0Ac91B79Ff7D2280d79e25f745960';// Deployed DealClient Contract address on calibration constcontractABI=contract.abi;// the path where the DealStatus.json is constcid='baga6ea4seaqpi75umesad5vlyzyf66vbzntoave4bebmkcqu4f6nq6rchhx3ckq'; // This handles proposing storage deals try{ const{ethereum}=window; if(ethereum) { constprovider=newethers.BrowserProvider(ethereum); constsigner=awaitprovider.getSigner(); dealStatus=newethers.Contract( contractAddress, contractABI, signer ); cid=newCID(commP) consttransaction=awaitdealStatus.submit(cid.bytes); constreceipt=awaittransaction.wait(); console.log(receipt); }else{ console.log("Ethereum object doesn't exist!"); } }catch(error) { console.log(error); return; } };

```

The full tutorial for uploading data using Lighthouse SDK and smart contract can be foundhere .

Manage storage deals with RaaS

RaaS (Replication, Renewal, and Repair as a Service) refers to the service provided for data stored in storage deals on the Filecoin network. When making storage deals with deal aggregators, such as lighthouse.storage, users have the option to register the RaaS job for the stored data. Subsequently, the aggregators monitor the status of the registered storage deals and initiate the necessary actions for replication, renewal, and repair as required.

When storing data using either the Lighthouse SDK or smart contracts, we can register a RaaS job.

- Lighthouse SDK: register replication, renew, and repair service by setting deal parameters when uploading data.
- Lighthouse smart contract: callingsubmitRaaS
- attaching RaaS parameters for the storage deal aggregation.
-

Ingredients

- Lighthouse.storage
-
    - SDK
-
    - : a JavaScript library that allows you to upload files to the Filecoin network.
-
    - smart contract
-
    - : solidity contract to submit and process storage deal aggregation requests.
- *
-

Instructions

1. register RaaS job when uploading with lighthouse SDK
2.

When uploading a file using the SDK, you have the flexibility to customize how it is stored in Lighthouse by adjusting the deal parameters.

- num_copies
- : Decide how many backup copies you want for your file. The Max limit is 3. For instance, if set to 3, your file will be stored by 3 different storage providers.
- repair_threshold
- : Determines when a storage sector is considered "broken" if a provider fails to confirm they still have your file. It's measured in "epochs", with 28800 epochs being roughly 10 days.
- renew_threshold
- : Specifies when your storage deal should be renewed. It's also measured in epochs.
- network
- : This should always be set to 'calibration' (for RAAS services to function) unless you want to use the mainnet.
-

```

Copy // Sample JSON of deal parameters constdealParams={ num_copies:2, repair_threshold:28800, renew_threshold:28800, network:'calibration', };

// register RaaS job with the aggregator SDK. constresponse=awaitlighthouse.upload(path,apiKey,false,dealParams);

```

1. register RaaS job when proposal storage deal using lighthouse smart contract
2.

Another way to register RaaS jobs is by interacting with the Lighthouse smart contract and submitting a CID of your choice to thesubmitRaaS function. This action creates a new deal request that will be picked up by the Lighthouse RaaS Worker, initiating the necessary replication, renewal, and repair processes.

```

Copy importcontractfrom"../contracts/DealStatus.json"; // ... other code constSubmitRaaS=async()=>{ constcontractAddress='0x01ccBC72B2f0Ac91B79Ff7D2280d79e25f745960';// Deployed DealClient Contract address constcontractABI=contract.abi;// the path where the DealStatus.json is constcid='baga6ea4seaqpi75umesad5vlyzyf66vbzntoave4bebmkcqu4f6nq6rchhx3ckq'; // This handles proposing storage deals try{ const{ethereum}=window; if(ethereum) { constprovider=newethers.BrowserProvider(ethereum); constsigner=awaitprovider.getSigner(); dealStatus=newethers.Contract( contractAddress, contractABI, signer ); cid=newCID(commP) consttransaction=awaitdealStatus.submitRaaS(cid.bytes,2,4,40); constreceipt=awaittransaction.wait(); console.log(receipt); }else{ console.log("Ethereum object doesn't exist!"); } }catch(error) { console.log(error); return; } };

```

Monitor storage deal status from a smart contract

TheDeal Bounty Contract also demonstrates a way to monitor the status of a Filecoin Storage Deal.

1. Import theMarketAPI .

```

Copy import{MarketAPI}from"../lib/filecoin-solidity/contracts/v0.8/MarketAPI.sol";

```

1. Use the MarketAPI functions to check the current status of a deal. An example is shown in claim_bounty():

```

Copy functionclaim_bounty(uint64deal_id)public{
MarketTypes.GetDealDataCommitmentReturnmemorycommitmentRet=MarketAPI.getDealDataCommitment(MarketTypes.GetDealDataCommitmentParams({id:deal_id}));
MarketTypes.GetDealProviderReturnmemoryproviderRet=MarketAPI.getDealProvider(MarketTypes.GetDealProviderParams({id:deal_id}));

authorizeData(commitmentRet.data,providerRet.provider,commitmentRet.size);

// get dealer (bounty hunter client) MarketTypes.GetDealClientReturnmemoryclientRet=MarketAPI.getDealClient(MarketTypes.GetDealClientParams({id:deal_id}));

// send reward to client send(clientRet.client); }

```

Incentivized data storage

There are two sides to incentivizing data onboarding –the first is to incentivize the client to upload data, which can be done with an ERC20 token included in a DataDAO that pays to wallets that upload data through the DataDAO. The second is to incentivize the storage providers to take a deal. Both are demonstrated in theDeal Bounty Contract .

Ingredients

- Foundry
- Solidity
- Filecoin Storage
- Filecoin Retrieval
-

Instructions

Note that the full solidity file for the Deal Bounty Contract can be foundHERE . This cookbook will pull relevant functions for you as a way to base your own code on.

1. The contract owner will deploy the contract, establishing the rules of the dataDAO.
2. Data pinners will add the deal CIDs intended to be incentivized to the list. This will allow storage providers to see which deals have additional incentives.
3.

```

Copy functionaddCID(bytes calldata cidraw,uint size)public{ require(msg.sender==owner); cidSet[cidraw]=true; cidSizes[cidraw]=size; }

```

1. The contract should then be funded by those who want to see the CID be accepted.
2.

```

Copy functioncall_actor_id(uint64method,uint256value,uint64flags,uint64codec,bytesmemoryparams,uint64id)publicreturns(bool,int256,uint64,bytesmemory) {
(boolsuccess,bytesmemorydata)=address(CALL_ACTOR_ID).delegatecall(abi.encode(method,value,flags,codec,params,id));
(int256exit,uint64return_codec,bytesmemoryreturn_value)=abi.decode(data,(int256,uint64,bytes)); return(success,exit,return_codec,return_value); }

// send 1 FIL to the filecoin actor at actor_id functionsend(uint64actorID)internal{ bytesmemoryemptyParams=""; deleteemptyParams;

uintoneFIL=1000000000000000000; HyperActor.call_actor_id(METHOD_SEND,oneFIL,DEFAULT_FLAG,Misc.NONE_CODEC,emptyParams,actorID);

}

```

1. Finally, the bounty is claimed by the storage providers that accepted the deal. This is done by using the MarketAPI to check the status of a deal.
2.

```

Copy functionclaim_bounty(uint64deal_id)public{
MarketTypes.GetDealDataCommitmentReturnmemorycommitmentRet=MarketAPI.getDealDataCommitment(MarketTypes.GetDealDataCommitmentParams({id:deal_id}));
MarketTypes.GetDealProviderReturnmemoryproviderRet=MarketAPI.getDealProvider(MarketTypes.GetDealProviderParams({id:deal_id}));

authorizeData(commitmentRet.data,providerRet.provider,commitmentRet.size);

// get dealer (bounty hunter client) MarketTypes.GetDealClientReturnmemoryclientRet=MarketAPI.getDealClient(MarketTypes.GetDealClientParams({id:deal_id}));

// send reward to client send(clientRet.client); }

```

Previous Data Storage Next Retrieve Data

Last updated1 month ago