@kleros/archon Documentation

Release 2.0.0

Sam Vitello

Oct 20, 2021

User Documentation

Archon is a javascript library written to make it easy to interact with `Arbitrable` and `Arbitrator` contacts. In particular, Archon can be used to take care of a lot of the hash validation work that is part of the Evidence Standard (ERC 1497).

Archon can be used with all Arbitrable and Arbitrator contracts that follow the ERC 792 standard and has the functionality to interact with all standardized methods.

To get up to speed on the standards please reference:

ERC 792 Arbitration Standard.

ERC 1497 Evidence Standard.

Tutorials on how to use ERC-792 and ERC-1497.

CHAPTER 1

Getting Started

1. Install Archon from NPM

Note: As of v2.0.0 `web3.js` is a peer-dependency and should be installed alongside `@kleros/archon`.

```
yarn add '@kleros/archon@^2.0.0' 'web3@^1.4.0'
```

2. Import Archon into your project

```
var Archon = require('@kleros/archon')
```

3. Initialize an instance of Archon using an Ethereum Provider and, optionally, an IPFS Gateway.

```
// By default Archon uses 'https://gateway.ipfs.io' as the default IPFS gateway. To␣
↪use your own gateway, pass the gateway URI as the 2nd parameter to Archon.
var archon = new Archon('https://mainnet.infura.io')
```

Note: A provider is needed so that the codebase knows how to connect to the Ethereum network. You can use a connection to your own node, or to a hosted node such as infura. `web3.js` will make RPC calls to the provider to fetch data from the blockchain.

Note: A gateway is needed to access `IPFS` files that might be referenced in evidence or metaEvidence.

Now you are all set! Use your *Archon* object to interact with *arbitrable* and *arbitrator* smart contracts on the blockchain.

CHAPTER 2

Hashing Examples

Here are some examples of evidence hashing and validation using archon. To see the full specification of Evidence and MetaEvidence see ERC 1497.

Note: All insignificant whitespace should be removed from JSON files before hashing. You can use `JSON.stringify` to remove whitespace.

2.1 Hash and Validate Hosted Evidence

In this example we will use an example evidence file hosted here: https://s3.us-east-2.amazonaws.com/kleros-examples/exampleEvidence.txt

```
var Archon = require('@kleros/archon');
var fs = require('fs');
var path = require('path');

// Bring in our evidence file downloaded from the link above to obtain the hash
var file = fs.readFileSync(path.resolve(__dirname, "./exampleEvidence.txt")).
↪toString();

// Hash the file using keccak-256
var evidenceHash = Archon.utils.multihashFile(
  file,
  0x1B // keccak-256
);

console.log(evidenceHash); //␣
↪Bce1WTQa7bfrJMFdEJuWV2xHsmj5JcDDyqBKGXu6PHZsn5e5oxkJ8cMJcuFDK1VsQYBtfrzgWkKCovWSvsacgN1XTj
```

(continues on next page)

(continued from previous page)

```
// Validate the hosted file against the hash we just produced.
Archon.utils.validateFileFromURI(
  'https://s3.us-east-2.amazonaws.com/kleros-examples/exampleEvidence.txt',
  { hash: evidenceHash }
).then(data => {
  console.log(data.isValid); // true
})
```

2.2 Hash and Validate Local Evidence

In this example we will use an example evidence file hosted here: https://s3.us-east-2.amazonaws.com/kleros-examples/exampleEvidence.txt

```
var Archon = require('@kleros/archon');
var fs = require('fs');
var path = require('path');

// Bring in our evidence file downloaded from the link above to obtain the hash
var file = fs.readFileSync(path.resolve(__dirname, "./exampleEvidence.txt")).
↪toString();

// Hash the file using keccak-256
var evidenceHash = Archon.utils.multihashFile(
  file,
  0x1B // keccak-256
);

console.log(evidenceHash); //␣
↪Bce1WTQa7bfrJMFdEJuWV2xHsmj5JcDDyqBKGXu6PHZsn5e5oxkJ8cMJcuFDK1VsQYBtfrzgWkKCovWSvsacgN1XTj

// Validate the original file against the hash we just produced.
console.log(Archon.utils.validMultihash(
  evidenceHash
  file,
); // true
```

2.3 Custom Hash Functions

If you would like to use a custom hashing function in your multihash you can pass one as an optional last parameter to `multihashFile` and `validMultihash` or as `option.customHashFn` to `validateFileFromURI`. For full documentation on these individual functions see *archon.utils*. You might want to do this in cases where you used a non-standard implementation of the hashing algorithm, or where there needs to be some data transformations before you apply the hashing algorithm, as is the case with *IPFS* hashes. This is only for the initial hashing algorithm in the multihash. Your hashing function should take a single `String` argument and return a `String` that is the hex representation of the hash.

1) Non standard hash function implementations

Note: You must still include a hashcode even if you are using a custom hash function. In order to have a valid multihash your hash must correspond to a valid hashcode. It is not recommended you use an unsupported hashing function that does not have a hashcode. If your hashes are not valid multihashes, consider validating your hashes outside of Archon.

Warning: If you use a custom hash function other interfaces may not be able to validate your hashes.

2.3.1 Example – Solidity keccak-256

Solidity uses a non standard implementation of the keccak-256 hashing algorithm. Therefore if we are using hashes produced by a smart contract we might need to validate using a custom hashing function.

Note: In this example the custom hash function is a non-standard implementation of keccak-256 so it can still use the hashcode 0x1B.

```
var Archon = require('@kleros/archon');
var Web3 = require('web3')

// Hash our "file" ('12345') using the solidity Sha3.
var nonStandardSha3Hash = Archon.utils.multihashFile(
  '12345',
  0x1B, // keccak-256
  Web3.utils.soliditySha3 // custom hash function
);

console.log(nonStandardSha3Hash); //␣
↪4ZqgPBxZrZkSXnTBm8G162mXVCNNbWrZa25CcvfGtQV1pRHH6X8vfLi89RKTA4c6tyfQsD5vzGvJozs24XvcLysiC3U6b

// Validate using the standard keccak-256 hashing algorithm.
console.log(Archon.utils.validMultihash(
  nonStandardSha3Hash,
  '12345'
)) // false

// Validate using the solidity sha3 implementation.
console.log(Archon.utils.validMultihash(
  nonStandardSha3Hash,
  '12345',
  Web3.utils.soliditySha3
)) // true
```

2.4 Hash Validation Troubleshooting

Here are some common mistakes that can cause your hashes to fail validation:

- Did not remove all insignificant whitespace before hashing JSON. This means all newlines and spaces in between your JSON values.

- Using a non-standard hash function. Out of the box, Archon supports these *hashing algorithms*. They are from the javascript library js-sha3. If you used a different hashing algorithm you will need to pass an implementation of it to Archon. See *Custom Hashing*.
- `sha2` vs `sha3`. Many libraries will specify their hashes are `sha256` without specifying if they are sha2 or sha3. `sha2-256` and `sha3-256` are different hashing algorithms and use different hashcodes.
- Not using the `base58` representation of the multihash hash. Multihash hashes can be expressed in many bases. Archon is expecting `base58` hashes.
- Did not include the original hash in some format to `validateFileFromURI`. Archon accepts hashes as the name of the file with no file type extension or using the property `selfHash` if the file is JSON. Otherwise if you have an alternate method of obtaining the hash pass it using `options.hash`.

2.5 Supported Hashing Algorithms and Hashcodes

Supported hashing algorithms:

Name	Multicode
sha3-512	0x14
sha3-384	0x15
sha3-256	0x16
sha3-224	0x17
keccak-224	0x1A
keccak-256	0x1B
keccak-384	0x1C
keccak-512	0x1D

Tip: By default, IPFS uses `sha2-256`. Many ethereum hashes are `keccak-256`.

Warning: Solidity uses a different implementation of the `keccak-256` algorithm. Hashes generated from smart contracts will need a `customHashFn` to verify.

Note: All insignificant whitespace should be removed from JSON files before hashing. You can use `JSON.stringify` to remove whitespace.

If a different hashing algorithm was used, pass it in the desired function with `customHashFn`. The function should expect a single string parameter.

A full list of possible algorithms and multicodes can be found here.

CHAPTER 3

IPFS Links and Validation

Many users will want to use decentralized file storage for their arbitrable DApps and will store `Evidence` and `MetaEvidence` on `IPFS`.

There are several things to keep in mind when using `IPFS` with `Archon`

3.1 IPFS Gateways

Most browsers do not currently support interacting with the `ipfs` network directly. Therefore, in order to fetch data from the `ipfs` network, we need to use an IPFS Gateway. Gateways are "standard" `http` URI's that return data from the `ipfs` network based on the ipfs path.

It is important that you choose a gateway that you trust, as a malicious gateway can return tampered data. Please heed the warning below:

> **Warning:** Archon considers data returned directly from a valid `ipfs` URI pre-validated. This is because hash validation is built into the protocol. As we need to use gateways to interact with the `ipfs` network at this time, *MAKE SURE YOU SET A GATEWAY THAT YOU TRUST*, or otherwise re-validate your `ipfs` hashes yourself.

The default gateway set is `https://gateway.ipfs.io`. This is the gateway provided by the Protocol Labs staff.

To set a custom gateway, use `archon.setIpfsGateway(uri)`. See documentation on setting a gateway *here*

3.2 IPFS URIs

`IPFS` URIs are recognized in any of these formats:

1) `/ipfs/Qm...../foo/bar`

2) `ipfs:/ipfs/Qm...../foo/bar`

3) `fs:/ipfs/Qm...../foo/bar`

3.3 Gateway URI's == Bad

It is not recommended that you link directly to an `IPFS` gateway as the `fileURI` in Evidence or MetaEvidence. It will not be able to be verified using Archon without a custom hashing function. The `IPFS` protocol does transformations on the data before hashing, therefore a hash can not be verified naively using the resulting `IPFS` multihash and the original file contents. In addition `IPFS` data should be able to be fetched by any gateway or node in the network, and a user should be able to use any gateway or node that they trust. Therefore hardcoding a gateway, which may not be trusted by the consumer, would require additional validation regardless.

Archon treats all `http(s)://` URI's the same, and will try to validate the hash as if they were standard hashes.

`Bad:`

```
const evidenceJSON = {
  fileURI: "https://gateway.ipfs.io/ipfs/
↪QmQhJRhvwgBPRdee18pK6QDECysFPuvuDLZMSeexdUfEiH"
}

archon.utils.validateFileFromURI(evidenceJSON.fileURI).then(data => {console.log(data.
↪isValid)})
> false
```

`Good:`

```
const evidenceJSON = {
  fileURI: "/ipfs/QmQhJRhvwgBPRdee18pK6QDECysFPuvuDLZMSeexdUfEiH"
}

archon.setIpfsGateway('https://gateway.ipfs.io')
archon.utils.validateFileFromURI(evidenceJSON.fileURI).then(data => {console.log(data.
↪isValid)})
> true
```

`Not Recommended:`

```
const ipfsHasher = data => {...}
const evidenceJSON = {
  fileURI: "https://gateway.ipfs.io/ipfs/
↪QmQhJRhvwgBPRdee18pK6QDECysFPuvuDLZMSeexdUfEiH"
}

archon.utils.validateFileFromURI(
  evidenceJSON.fileURI,
  { customHashFn: ipfsHasher }
).then(data => {
  console.log(data.isValid)
})
> true
```

CHAPTER 4

Evidence Examples

Here are some examples of how you can use Archon to submit and retrieve evidence. In these examples we will assume we are using an `Arbitrable` contract that has a function:

```
function submitEvidence(string _evidence) public {
    ...
    emit Evidence(_arbitrator, _disputeID, msg.sender, _evidence);
}
```

4.1 Submit and Fetch Evidence

This example demonstrates how to submit evidence, as per the Evidence Standard, and how to retrieve the Evidence using `Archon`.

In this example we will use an example evidence file hosted here: https://s3.us-east-2.amazonaws.com/kleros-examples/exampleEvidence.txt

4.1.1 Part 1: Create Evidence JSON File

```
var Archon = require('@kleros/archon');
var fs = require('fs');
var path = require('path');

// initialize Archon. By default it uses IPFS gateway https://gateway.ipfs.io
var archon = new Archon("https://mainnet.infura.io");

// First we need the hash of our evidence file. Download the file and hash it.
var file = fs.readFileSync(path.resolve(__dirname, "./exampleEvidence.txt")).
↪toString();
```

(continues on next page)

(continued from previous page)

```
var evidenceHash = archon.utils.multihashFile(
  file,
  0x1B // keccak-256
);

console.log(evidenceHash); //␣
↪Bce1WTQa7bfrJMFdEJuWV2xHsmj5JcDDyqBKGXu6PHZsn5e5oxkJ8cMJcuFDK1VsQYBtfrzgWkKCovWSvsacgN1XTj

// Now we can construct our EvidenceJSON from the Evidence Standard
var evidenceJSON = {
  fileURI: "https://s3.us-east-2.amazonaws.com/kleros-examples/exampleEvidence.txt",
  fileHash: evidenceHash,
  name: "Example Evidence",
  description: "This evidence shows how to properly utilize hashing and Archon to␣
↪submit valid evidence!"
}

var evidenceJSONHash = archon.utils.multihashFile(
  evidenceJSON,
  0x1B // keccak-256
)

console.log(evidenceJSONHash) //

// Write our JSON to a file so we can host it with IPFS
fs.writeFileSync(path.resolve(__dirname, `./${evidenceJSONHash}`), JSON.
↪stringify(evidenceJSON));
```

4.1.2 Part 2: Host with IPFS and get the hash

Note: If it is not hosted via IPFS make sure to use the multihash as the suffix of the URI or include `selfHash` in the `JSON`.

```
ipfs add -r exampleEvidence.json
> added QmdBNTwDazHsYXk9xW9JnM4iVGpdUnZni1DS4pyF3adKq1 exampleEvidence.json
```

4.1.3 Part 2 (Alternate): Host on any cloud provider (e.g. AWS) using hash as file-name

Note: If it is not hosted via IPFS make sure to use the multihash as the suffix of the URI or include `selfHash` in the `JSON`.

```
# Add file to aws
aws s3 cp ./<hash> s3://kleros-examples/<hash>

# file can be found at https://s3.us-east-2.amazonaws.com/kleros-examples/<hash>
```

4.1.4 Part 3: Submit the evidence

```
var Web3 = require("web3");
// You will need to submit your transaction from a node or wallet that has funds to␣
↪pay gas fees.
var web3 = new Web3("https://mainnet.infura.io");

// Load the arbitrable contract to submit our evidence
// See web3 docs for more information on interacting with your contract
var contractInstance = new web3.eth.Contract(<My Contract ABI>, <My Contract Address>
↪);

// Call submit evidence using the IPFS hash from our JSON file
contractInstance.methods.submitEvidence(
  '/ipfs/QmdBNTwDazHsYXk9xW9JnM4iVGpdUnZni1DS4pyF3adKq1'
).send({
  from: "0x54FcB2536b3E1DD222aD2c644535244000b377cd",
  gas: 500000
});

// OR

// Call submit evidence using the hosted URI
contractInstance.methods.submitEvidence(
  'https://s3.us-east-2.amazonaws.com/kleros-examples/<hash>'
).send({
  from: "0x54FcB2536b3E1DD222aD2c644535244000b377cd",
  gas: 500000
});
```

4.1.5 Part 4: Retrieve Evidence from the contract

```
var Archon = require("@kleros/archon");
var archon = new Archon("https://mainnet.infura.io");

archon.arbitrable.getEvidence(
  <My Contract Address>,
  <Arbitrator Address>,
  <Dispute ID>
).then(evidence => {
  console.log(evidence)
});

> [{
  evidenceJSON: {
    fileURI: "https://s3.us-east-2.amazonaws.com/kleros-examples/exampleEvidence.txt",
    fileHash:
↪"Bce1WTQa7bfrJMFdEJuWV2xHsmj5JcDDyqBKGXu6PHZsn5e5oxkJ8cMJcuFDK1VsQYBtfrzgWkKCovWSvsacgN1XTj
↪",
    name: "Example Evidence",
    Description: "This evidence shows how to properly utilize hashing and Archon to␣
↪submit valid evidence!"
  },
  evidenceValid: true,
  fileValid: true,
```

(continues on next page)

(continued from previous page)

```
  submittedBy: <My Account>,
  submittedAt: <Timestamp>
}]
```

4.2 Fetch MetaEvidence for Dispute

This example demonstrates how to retrieve the MetaEvidence for a dispute using `Archon`.

```
var Archon = require('@kleros/archon');

// initialize Archon. By default it uses IPFS gateway https://gateway.ipfs.io
var archon = new Archon("https://mainnet.infura.io");

// Fetch the event log emitted by the Arbitrable contract when a dispute is raised
archon.arbitrable.getDispute(
  "0x91697c78d48e9c83b71727ddd41ccdc95bb2f012", // arbitrable contract address
  "0x211f01e59b425253c0a0e9a7bf612605b42ce82c", // arbitrator contract address
  23 // dispute unique identifier
).then(disputeLog => {
  // use the metaEvidenceID to fetch the MetaEvidence event log
  archon.arbitrable.getMetaEvidence(
    "0x91697c78d48e9c83b71727ddd41ccdc95bb2f012", // arbitrable contract address
    disputeLog.metaEvidenceID
  ).then(metaEvidenceData => {
    console.log(metaEvidenceData)
  })
})

> {
  metaEvidenceValid: true,
  fileValid: true,
  interfaceValid: true,
  metaEvidenceJSON: {"fileURI": "/ipfs/...", ...},
  submittedAt: 1539025000,
  blockNumber: 6503570,
  transactionHash: "0x340fdc6e32ef24eb14f9ccbd2ec614a8d0c7121e8d53f574529008f468481990
↪"
}
```

CHAPTER 5

MetaEvidence Dynamic Scripts

Warning: Node.js implementation of the script sandbox is incomplete. dynamic scripts can currently only be evaluated in the browser.

dynamicScriptURI as specified in ERC 1497. allow for metaEvidence to be changed dynamically.

In order to support asynchronous scripts Archon provides global functions `resolveScript` and `rejectScript`. Please use these when you are returning the updates to the metaEvidence

5.1 Available Global Variables

When you create a script you can assume that you will have global access to these variables:

1) `resolveScript` - `function`: Call this function to return the result of your script.
2) `rejectScript` - `function`: To throw a handled error in your script you can call scriptRejecter.
3) `scriptParameters` - `object`: These are the parameters you pass as *options["scriptParameters"]* in getMetaEvidence

5.1.1 Example

```
/* Script that is hosted at "/ipfs/..."
  "
    function MetaEvideceFetcher () {
      const disputeID = scriptParameters.disputeID

      ... async code to lookup data here ...

      if (data) {
        resolveScript({
```

(continues on next page)

(continued from previous page)

```
          newMetaEvidenceParam: "Look at me!"
        })
      } else {
        rejectScript("Unable to fetch data")
      }
    }

    MetaEvideceFetcher()
  "
*/

archon.arbitrable.getMetaEvidence(
  '0x91697c78d48e9c83b71727ddd41ccdc95bb2f012',
  1,
  {
    scriptParameters: {
      disputeID: 2
    }
  }
).then(data => {
  console.log(data)
})

> {
    metaEvidenceValid: true,
    fileValid: true,
    interfaceValid: false,
    dynamicScriptURI: "/ipfs/...",
    metaEvidenceJSON: {
      "fileURI": "/ipfs/...",
      "newMetaEvidenceParam": "Look at me!",
      ...
    },
    submittedAt: 1539025000,
    blockNumber: 6503570,
    transactionHash:
↪"0x340fdc6e32ef24eb14f9ccbd2ec614a8d0c7121e8d53f574529008f468481990"
  }
```

CHAPTER 6

Archon

Class

This is the main class of Archon.

```
var Archon = require('@kleros/archon');

> Archon.modules
> Archon.utils
> Archon.version
```

6.1 Archon.modules

Property of Archon class

```
Archon.modules
```

Will return an object with the classes of all major sub modules, to be able to instantiate them manually.

6.1.1 Returns

`Object`: A list of modules:

- `Arbitrator` - `Function`: The module for interacting with Arbitrator contracts. See *archon.arbitrator* for more.
- `Arbitrable` - `Function`: The module for interacting with Arbitrable contracts. See *archon.arbitrable* for more.

6.1.2 Example

```
Archon.modules
> {
    Arbitrator: Arbitrator function(provider),
    Arbitrable: Arbitrable function(provider)
}
```

6.2 Archon.version

Property of Archon class and instance of archon

```
Archon.version
archon.version
```

Contains the version of the `archon` container object.

6.2.1 Returns

`String`: The current version.

6.2.2 Example

```
archon.version;
> "0.1.0"
```

6.3 Archon.utils

Property of Archon class and instance of archon

```
Archon.utils
archon.utils
```

Utility functions are also exposes on the `Archon` class object directly.

See *archon.utils* for more.

6.4 new Archon()

```
new Archon(ethereumProvider, ipfsGatewayURI='https://gateway.ipfs.io')
```

6.4.1 Parameters

1) `ethereumProvider` - `String|Object`: The provider object or URI of the Ethereum provider.
2) `ipfsGatewayURI` - `String`: The URI of a trusted `IPFS` gateway for fetching files from the IPFS network.

6.4.2 Example

Instantiate Archon as an object to have access to all initialized modules.

```
var Archon = require('@kleros/archon');

// "Web3.providers.givenProvider" will be set if in an Ethereum supported browser.
var archon = new Archon('ws://some.local-or-remote.node:8546');

> archon.arbitrator
> archon.arbitrable
> archon.utils
> archon.version
```

6.5 archon.setProvider()

```
archon.setProvider(myProvider)
archon.arbitrable.setProvider(myProvider)
archon.arbitrator.setProvider(myProvider)
...
```

Will change the ethereum provider.

Note: If called on the `archon` class it will update the provider for all submodules. `archon.arbitrable`, `archon.arbitrator`, etc.

6.5.1 Parameters

1. `myProvider` - `Object|String`: A provider object or URI.

6.5.2 Example

```
var Archon = require('archon');
var archon = new Archon('http://localhost:8545');

// change provider for all submodules
archon.setProvider('ws://localhost:8546');
// or
archon.setProvider(new Web3.providers.WebsocketProvider('ws://localhost:8546'));

// change provider for arbitrator
archon.arbitrator.setProvider('https://mainnet.infura.io/')
```

6.6 archon.setIpfsGateway()

```
archon.setIpfsGateway(ipfsGatewayURI)
```

Will change the `IPFS` gateway used to fetch and validate data.

6.6.1 Parameters

1. `ipfsGatewayURI` - `Object`: A URI to a trusted IPFS gateway .

6.6.2 Example

```
var Archon = require('archon');
var archon = new Archon('http://localhost:8545');

// change IPFS gateway
archon.setIpfsGateway('https://cloudflare-ipfs.com/');
```

CHAPTER 7

archon.arbitrable

This package provides the functionality to interact with `Arbitrable` smart contracts. An arbitrable contract creates a dispute and enforces the ruling made by an arbitrator. Arbitrable contracts are also responsible for `Evidence` and `MetaEvidence`.

Tip: See ERC 792. for more information on `Arbitrable` contracts.

7.1 getEvidence()

```
archon.arbitrable.getEvidence(contractAddress, arbitratorAddress, evidenceGroupID,␣
↪options={});
```

Fetch and validate evidence via the Arbitrable smart contract `Evidence` event logs. For a particular dispute.

Tip: See ERC 1497. for the `EvidenceJSON` spec and information on how to correctly use the events with hashes

7.1.1 Parameters

1) `contractAddress` - `String`: The address of the arbitrable contract.
2) `arbitratorAddress` - `String`: The address of the arbitrator contract.
3) `evidenceGroupID` - `Number`: The evidence group ID.
4) `options` - `Object`: Optional parameters.

The options parameter can include:

Key	Type	Description
strict	bool	If true, an error will throw if hash or chain ID validations fail.
strictHashes	bool	[DEPRECATED] If true, an error will throw if hash validations fail.
customHashFn	fn	Hashing function that should be used to validate the hashes.
fromBlock	int	The block where we start searching for event logs.
toBlock	int	The block where we will stop searching for event logs.
filters	object	Additional filters for event logs.

Tip: Use *getDispute* to get the evidenceGroupID for a dispute.

7.1.2 Returns

`Promise.<Object[]>` - A Promise resolving to an array of objects containing the `EvidenceJSON`, the validity of the `JSON` and evidence file, and submission information.

```
{
  evidenceJSONValid: <Bool>, // validity of evidenceJSON
  fileValid: <Bool>, // validity of evidence found at evidenceJSON.fileURI
  evidenceJSON: <Object>,
  submittedBy: <String>, // from event log
  submittedAt: <Number>, // epoch timestamp in seconds
  blockNumber: <Number>,
  transactionHash: <String> // The hash of the submission transaction
}
```

7.1.3 Example

```
archon.arbitrable.getEvidence(
  "0x91697c78d48e9c83b71727ddd41ccdc95bb2f012", // arbitrable contract address
  "0x211f01e59b425253c0a0e9a7bf612605b42ce82c", // arbitrator contract address
  1, // dispute ID
  {
    strict: true
  }
).then(data => {
  console.log(data)
})

> [{
    evidenceJSONValid: true,
    fileValid: true,
    evidenceJSON: {"fileURI": "/ipfs/...", ...},
    submittedBy: "0x8254175f6a6E0FE1f63e0eeb0ae487cCf3950BFb",
    submittedAt: 1539022733,
    blockNumber: 6503576,
    transactionHash:
↪"0xe91603b9d4bf506972820f499bf221cdfb48cbfd426125af5ab647dca39a3f4e"
  },
  {
```

(continues on next page)

(continued from previous page)

```
    evidenceJSONValid: true,
    fileValid: true,
    evidenceJSON: {"fileURI": "/ipfs/...", ...},
    submittedBy: "0xc55a13e36d93371a5b036a21d913a31CD2804ba4",
    submittedAt: 1539025000,
    blockNumber: 6503570,
    transactionHash:
↪"0x340fdc6e32ef24eb14f9ccbd2ec614a8d0c7121e8d53f574529008f468481990"
  }]
```

7.2 getMetaEvidence()

```
archon.arbitrable.getMetaEvidence(contractAddress, metaEvidenceID, options={});
```

Fetch and validate MetaEvidence via the Arbitrable smart contract `MetaEvidence` event logs. There are up to 3 hashes validated. The hash of the `MetaEvidenceJSON`, the hash of the primary document file specified at `MetaEvidenceJSON.fileURI`, and the hash of the external interface used to render the evidence found at `MetaEvidenceJSON.evidenceDisplayInterfaceURL`.

Tip: See ERC 1497. for the `MetaEvidenceJSON` spec and information on how to correctly use the events with hashes

7.2.1 Parameters

1) `contractAddress` - `String`: The address of the arbitrable contract.
2) `metaEvidenceID` - `Number|String`: The unique identifier of the MetaEvidence event log.
3) `options` - `Object`: Optional parameters.

The options parameter can include:

Tip: Use *getDispute* to get the metaEvidenceID for a dispute.

Note: If more than one MetaEvidence exists for the given metaEvidenceID, only the first submitted metaEvidence will be returned.

Tip: See *MetaEvidece Scripts* for detailed description on creating scripts compatible with Archon.

7.2.2 Returns

`Promise.<Object>` - Promise resolving to an object containing the `MetaEvidenceJSON` and the validity of the the hashes

```
{
  metaEvidenceValid: <Bool>,
  fileValid: <Bool>,
  interfaceValid: <Bool>,
  metaEvidenceJSON: <Object>,
  submittedAt: <Number>,
  blockNumber: <Number>,
  transactionHash: <String>
}
```

7.2.3 Example

```
archon.arbitrable.getMetaEvidence(
  '0x91697c78d48e9c83b71727ddd41ccdc95bb2f012',
  1,
  {
    strict: false
  }
).then(data => {
  console.log(data)
})

> {
    metaEvidenceValid: true,
    fileValid: true,
    interfaceValid: false,
    metaEvidenceJSON: {"fileURI": "/ipfs/...", ...},
    submittedAt: 1539025000,
    blockNumber: 6503570,
    transactionHash:
↪"0x340fdc6e32ef24eb14f9ccbd2ec614a8d0c7121e8d53f574529008f468481990"
  }
```

7.3 getRuling()

```
archon.arbitrable.getRuling(contractAddress, arbitratorAddress, disputeID, options={}
↪);
```

Fetch data from the `Ruling` event log.

7.3.1 Parameters

1) `contractAddress` - `String`: The address of the arbitrable contract.
2) `arbitratorAddress` - `String`: The address of the arbitrator contract.
3) `disputeID` - `Number`: The unique identifier of the dispute.
4) `options` - `Object`: Optional parameters.

The options parameter can include:

Key	Type	Description
fromBlock	int	The block where we start searching for event logs.
toBlock	int	The block where we will stop searching for event logs.
filters	object	Additional filters for event logs.

7.3.2 Returns

`Promise.<Object>` - A Promise resolving to data from the ruling event log, including the final ruling.

```
{
  ruling: <String>, // The ruling returned as a number string
  ruledAt: <Number>, // epoch timestamp in seconds
  blockNumber: <Number>,
  transactionHash: <String> // The hash of the submission transaction
}
```

7.3.3 Example

```
archon.arbitrable.getRuling(
  '0x91697c78d48e9c83b71727ddd41ccdc95bb2f012',
  '0x211f01e59b425253c0a0e9a7bf612605b42ce82c',
  1
).then(data => {
  console.log(data)
})

> {
  ruling: "1",
  ruledAt: 1539025000,
  blockNumber: 6503570,
  transactionHash: "0x340fdc6e32ef24eb14f9ccbd2ec614a8d0c7121e8d53f574529008f468481990
↪"
}
```

7.4 getDispute()

```
archon.arbitrable.getDispute(contractAddress, arbitratorAddress, disputeID, options={}
↪);
```

Fetch the dispute creation event. This event is used to link `metaEvidenceID` to a dispute.

7.4.1 Parameters

1) `contractAddress` - `String`: The address of the arbitrable contract.
2) `arbitratorAddress` - `String`: The address of the arbitrator contract.

3) `disputeID` - `Number`: The unique identifier of the dispute.

4) `options` - `Object`: Optional parameters.

The options parameter can include:

Key	Type	Description
fromBlock	int	The block where we start searching for event logs.
toBlock	int	The block where we will stop searching for event logs.
filters	object	Additional filters for event logs.

7.4.2 Returns

`Promise.<Object>` - A Promise resolving to data from the dispute event log including the `metaEvidenceID`

```
{
  metaEvidenceID: <String>,
  evidenceGroupID: <String>,
  createdAt: <Number>,
  blockNumber: <Number>,
  transactionHash: <String>
}
```

7.4.3 Example

```
archon.arbitrable.getDispute(
  '0x91697c78d48e9c83b71727ddd41ccdc95bb2f012',
  '0x211f01e59b425253c0a0e9a7bf612605b42ce82c',
  1
).then(data => {
  console.log(data)
})

> {
  metaEvidenceID: "0",
  evidenceGroupID: "3",
  createdAt: 1539025000,
  blockNumber: 6503570,
  transactionHash: "0x340fdc6e32ef24eb14f9ccbd2ec614a8d0c7121e8d53f574529008f468481990
↪"
}
```

CHAPTER 8

archon.arbitrator

This package provides the functionality to interact with Arbitrator contracts. An `Arbitrator` contract makes rulings on disputes.

Tip: See ERC 792. for more information on `Arbitrator` contracts.

8.1 DisputeStatus

The dispute status enum used to map a `DisputeStatus` int to plain text status.

The enum is defined in the `Arbitrator` contract as follows:

```
enum DisputeStatus {Waiting, Appealable, Solved}
```

8.1.1 Example

```
archon.arbitrator.DisputeStatus[1]
> 'Appealable'
```

8.2 getArbitrationCost()

```
archon.arbitrator.getArbitrationCost(contractAddress, extraData='0x0');
```

Get the arbitration cost based from `extraData`.

Note: The format of extraData will depend on the implementation of the Arbitrator.

8.2.1 Parameters

1) `contractAddress` - `String`: The address of the arbitrator contract.
2) `extraData` - `String`: Hex string representing some bytes used by arbitrator for customization of dispute resolution.

8.2.2 Returns

`Promise.<String>` - Promise resolves to the cost of arbitration (in WEI)

8.2.3 Example

```
archon.arbitrator.getArbitrationCost(
  '0x211f01e59b425253c0a0e9a7bf612605b42ce82c'
).then(data => {
  console.log(data)
})
> "150000000000000000"
```

8.3 getAppealCost()

```
archon.arbitrator.getAppealCost(contractAddress, disputeID, extraData='0x0');
```

Get the cost of an appeal for a specific dispute base on `extraData`.

Note: The format of extraData will depend on the implementation of the Arbitrator.

8.3.1 Parameters

1) `contractAddress` - `String`: The address of the arbitrator contract.
2) `disputeID` - `Number`: The unique identifier of the dispute.
3) `extraData` - `String`: Hex string representing some bytes used by arbitrator for customization of dispute resolution.

8.3.2 Returns

`Promise.<String>` - Promise resolves to the cost of appeal (in WEI)

8.3.3 Example

```
archon.arbitrator.getAppealCost(
  '0x211f01e59b425253c0a0e9a7bf612605b42ce82c'
).then(data => {
  console.log(data)
})
> "150000000000000000"
```

8.4 getCurrentRuling()

```
archon.arbitrator.getCurrentRuling(contractAddress, disputeID);
```

Get the current ruling of a dispute.

8.4.1 Parameters

1) `contractAddress` - `String`: The address of the arbitrator contract.
2) `disputeID` - `Number`: The unique identifier of the dispute.

8.4.2 Returns

`Promise.<String>` - Promise resolves to the current ruling of the dispute.

8.4.3 Example

```
archon.arbitrator.getCurrentRuling(
  '0x211f01e59b425253c0a0e9a7bf612605b42ce82c',
  15
).then(data => {
  console.log(data)
})
> "2"
```

8.5 getDisputeStatus()

```
archon.arbitrator.getDisputeStatus(contractAddress, disputeID);
```

Get the status of the dispute.

Tip: Use archon.arbitrator.DisputeStatus to get a plain text status.

8.5.1 Parameters

1) `contractAddress` - `String`: The address of the arbitrator contract.
2) `disputeID` - `Number`: The unique identifier of the dispute.

8.5.2 Returns

`Promise.<String>` - Promise resolves to the status of a dispute as a number string.

8.5.3 Example

```
archon.arbitrator.getDisputeStatus(
  '0x211f01e59b425253c0a0e9a7bf612605b42ce82c',
  15
).then(data => {
  console.log(data)
})
> "0"
```

8.6 getDisputeCreation()

```
archon.arbitrator.getDisputeCreation(contractAddress, disputeID, options={})
```

Fetch the dispute creation event log and return data about the dispute creation.

8.6.1 Parameters

1) `contractAddress` - `String`: The address of the arbitrator contract.
2) `disputeID` - `Number`: The unique identifier of the dispute.
3) `options` - `Object`: Optional parameters.

The options parameter can include:

Key	Type	Description
fromBlock	int	The block where we start searching for event logs.
toBlock	int	The block where we will stop searching for event logs.
filters	object	Additional filters for event logs.

8.6.2 Returns

`Promise.<Object>` - Promise resolves to an object with data from the dispute creation log.

```
{
  createdAt: <Number>, // epoch timestamp in seconds
  arbitrableContract: <String>,
  blockNumber: <Number>,
  transactionHash: <String>
}
```

8.6.3 Example

```
archon.arbitrator.getDisputeCreation(
  '0x211f01e59b425253c0a0e9a7bf612605b42ce82c',
  15
).then(data => {
  console.log(data)
})
> {
  createdAt: 1539000000,
  arbitrableContract: "0x91697c78d48e9c83b71727ddd41ccdc95bb2f012",
  blockNumber: 6459000,
  transactionHash: "0x340fdc6e32ef24eb14f9ccbd2ec614a8d0c7121e8d53f574529008f468481990
↪"
}
```

8.7 getAppealDecision()

```
archon.arbitrator.getAppealDecision(contractAddress, disputeID, appealNumber, options=
↪{})
```

Fetch the appeal decision event log and return data about the appeal.

8.7.1 Parameters

1) `contractAddress` - `String`: The address of the arbitrator contract.
2) `disputeID` - `Number`: The unique identifier of the dispute.
3) `appealNumber` - `Number`: The appeal number. Must be >= 1
4) `options` - `Object`: Optional parameters.

The options parameter can include:

Key	Type	Description
fromBlock	int	The block where we start searching for event logs.
toBlock	int	The block where we will stop searching for event logs.
filters	object	Additional filters for event logs.

8.7.2 Returns

`Promise.<Object>` - Promise resolves to an object with data from the appeal decision log.

```
{
  appealedAt: <Number>, // epoch timestamp in seconds
  arbitrableContract: <String>,
  blockNumber: <Number>,
  transactionHash: <String>
}
```

8.7.3 Example

```
archon.arbitrator.getAppealDecision(
  '0x211f01e59b425253c0a0e9a7bf612605b42ce82c',
  15,
  1
).then(data => {
  console.log(data)
})
> {
  appealedAt: 1539025733,
  arbitrableContract: "0x91697c78d48e9c83b71727ddd41ccdc95bb2f012",
  blockNumber: 6459276,
  transactionHash: "0x340fdc6e32ef24eb14f9ccbd2ec614a8d0c7121e8d53f574529008f468481990
↪"
}
```

CHAPTER 9

archon.utils

This package provides utility functions that can be used to validate hashes of `Evidence` and `MetaEvidence`

Note: If using `/ipfs/...` URIs the the IPFS Gateway will be used to fetch and validate the data.

Warning: It is not recommended that you link directly to an `IPFS` gateway in the evidence because it will not be able to be verified without a custom hashing function. The `IPFS` protocol does transformations on the data before hashing, therefore a hash can not be verified naively using the resulting `IPFS` multihash and the original file contents.

9.1 validateFileFromURI()

```
archon.utils.validateFileFromURI(fileURI, options={});
```

Takes a the URI of the file and validates the hash. In order for to validate the hash, the original `multihash` must be included as:

1. The suffix of the URI (e.g. `https://file-uri/QmUQMJbfiQYX7k6SWt8xMpR7g4vwtAYY1BTeJ8UY8JWRs9`)
2. As a member of the `options` parameter (e.g. `options.hash = 'QmUQMJbfiQYX7k6SWt8xMpR7g4vwtAYY1BTeJ8UY8JWRs9'`)
3. In the resulting file JSON using the key `selfHash`

9.1.1 Parameters

1) `fileURI` - `String`: The URI where the file can be fetched. Currently only support protocols: `http(s)://`, `/ipfs/`

2) `options` - `Object`: Optional parameters.

The options parameter can include:

Key	Type	Description
preValidated	bool	If file has been pre-validated this will just fetch file and set isValid = true.
hash	string	The original hash to compare the file against.
strict	bool	If true, an error will throw if hash or chain ID validations fail.
strictHashes	bool	[DEPRECATED] If true, an error will throw if hash validations fail.
customHashFn	fn	A custom hash function to use to validate the file.

9.1.2 Returns

`Promise.<Object>` - Promise that resolves to an object containing the file as well as if the file is valid

```
{
  file: <String|Object>,
  isValid: <Bool>
}
```

9.1.3 Example

```
archon.utils.validateFileFromURI(
  'https://s3.us-east-2.amazonaws.com/kleros-examples/exampleEvidence.txt',
  {
    hash:
↪'Bce1WTQa7bfrJMFdEJuWV2xHsmj5JcDDyqBKGXu6PHZsn5e5oxkJ8cMJcuFDK1VsQYBtfrzgWkKCovWSvsacgN1XTj
↪'
  }
).then(data => {
  console.log(data)
})
> {
  file: 'This is an example evidence file. Here we could have...',
  isValid: true
}
```

9.2 validMultihash()

```
archon.utils.validMultihash(multihashHex, file, customHashFn=null);
```

Verify if the `multihash` of a file matches the file contents.

Note: Hashes should be base58 encoded `Strings`

Supported hashing algorithms:

Name	Multicode
sha3-512	0x14
sha3-384	0x15
sha3-256	0x16
sha3-224	0x17
keccak-224	0x1A
keccak-256	0x1B
keccak-384	0x1C
keccak-512	0x1D

Tip: By default, IPFS uses `sha2-256`. Many ethereum hashes are `keccak-256`.

Warning: Solidity uses a different implementation of the `keccak-256` algorithm. Hashes generated from smart contracts will need a `customHashFn` to verify.

Note: All insignificant whitespace should be removed from JSON files before hashing. You can use `JSON.stringify` to remove whitespace.

If a different hashing algorithm was used, pass it in the desired function with `customHashFn`. The function should expect a single string parameter.

A full list of possible algorithms and multicodes can be found here.

9.2.1 Parameters

1) `multihashHex` - `String`: The base58 multihash hex string.
2) `file` - `Object|String`: The raw File or JSON object we are verifying the hash of.
3) `customHashFn` - `Function`: `<optional>` A custom hashing algorithm used to generate original hash.

9.2.2 Returns

`Bool` - If the provided hash and file are valid.

9.2.3 Example

```
archon.utils.validMultihash(

↪'Bce1WTQa7bfrJMFdEJuWV2xHsmj5JcDDyqBKGXu6PHZsn5e5oxkJ8cMJcuFDK1VsQYBtfrzgWkKCovWSvsacgN1XTj
↪',
  'This is an example evidence file. Here we could have some document...'
)
> true
```

9.3 multihashFile()

```
archon.utils.multihashFile(file, multicode, customHashFn=null);
```

Generate the base58 multihash hex of a file

Supported hashing algorithms:

Name	Multicode
sha3-512	0x14
sha3-384	0x15
sha3-256	0x16
sha3-224	0x17
keccak-224	0x1A
keccak-256	0x1B
keccak-384	0x1C
keccak-512	0x1D

Tip: By default, IPFS uses `sha2-256`. Many ethereum hashes are `keccak-256`.

Warning: Solidity uses a different implementation of the `keccak-256` algorithm. Hashes generated from smart contracts will need a `customHashFn` to verify.

Note: All insignificant whitespace should be removed from JSON files before hashing. You can use `JSON.stringify` to remove whitespace.

If a different hashing algorithm was used, pass it in the desired function with `customHashFn`. The function should expect a single string parameter.

A full list of possible algorithms and multicodes can be found here.

9.3.1 Parameters

1) `file` - `Object|String`: The raw File or JSON object to hash
2) `multicode` - `Number`: The multihash hashing algorithm identifier.
3) `customHashFn` - `Function`: `<optional>` A custom hashing algorithm used to generate the hash.

9.3.2 Returns

`String` - base58 multihash of file.

9.3.3 Example

```
archon.utils.multihashFile(
  'This is an example evidence file. Here we could have some document...',
  0x1B // 27 => keccak-256
)
>
↪"Bce1WTQa7bfrJMFdEJuWV2xHsmj5JcDDyqBKGXu6PHZsn5e5oxkJ8cMJcuFDK1VsQYBtfrzgWkKCovWSvsacgN1XTj
↪"
```