
hyperdb

Distributed scalable database.

```
1 npm install hyperdb
```

Read ARCHITECTURE.md for details on how hyperdb works.

Usage

```
1 var hyperdb = require('hyperdb')
2
3 var db = hyperdb('./my.db', {valueEncoding: 'utf-8'})
4
5 db.put('/hello', 'world', function (err) {
6   if (err) throw err
7   db.get('/hello', function (err, nodes) {
8     if (err) throw err
9     console.log('/hello --> ' + nodes[0].value)
10  })
11 })
```

API

var db = hyperdb(storage, [key], [options]) Create a new hyperdb.

storage can be a string or a function. If a string like the above example, the random-access-file storage module is used; the resulting folder with the data will be whatever **storage** is set to.

If **storage** is a function, it will be called with every filename hyperdb needs to operate on. There are many providers for the abstract-random-access interface. e.g.

```
1 var ram = require('random-access-memory')
2 var feed = hyperdb(function (filename) {
3   // filename will be one of: data, bitfield, tree, signatures, key,
4   //   secret_key
5   // the data file will contain all your data concattenated.
6   // just store all files in ram by returning a random-access-memory
7   //   instance
8   return ram()
9 })
```

key is a **Buffer** containing the local feed's public key. If you do not set this the public key will be loaded from storage. If no key exists a new key pair will be generated.

Options include:

```
1 {
2   map: node => mappedNode, // map nodes before returning them
3   reduce: (a, b) => someNode, // reduce the nodes array before
    returning it
4   firstNode: false, // set to true to reduce the nodes array to the
    first node in it
5   valueEncoding: 'binary' // set the value encoding of the db
6 }
```

db.key Buffer containing the public key identifying this hyperdb.

Populated after `ready` has been emitted. May be `null` before the event.

db.discoveryKey Buffer containing a key derived from the `db.key`. In contrast to `db.key` this key does not allow you to verify the data but can be used to announce or look for peers that are sharing the same hyperdb, without leaking the hyperdb key.

Populated after `ready` has been emitted. May be `null` before the event.

db.on('ready') Emitted exactly once: when the db is fully ready and all static properties have been set. You do not need to wait for this when calling any async functions.

db.version(callback) Get the current version identifier as a buffer for the db.

var checkout = db.checkout(version) Checkout the db at an older version. The checkout is a DB instance as well. Version should be a version identifier returned by the `db.version` api or an array of nodes returned from `db.heads`.

db.put(key, value, [callback]) Insert a new value. Will merge any previous values seen for this key.

db.get(key, callback) Lookup a string `key`. Returns a nodes array with the current values for this key. If there is no current conflicts for this key the array will only contain a single node.

db.del(key, callback) Delete a string `key`.

db.batch(batch, [callback]) Insert a batch of values efficiently, in a single atomic transaction. A batch should be an array of objects that look like this:

```
1 {
2   type: 'put',
3   key: someKey,
4   value: someValue
5 }
```

`callback`'s parameters are `err`, `nodes`, where `nodes` is an array of the batched nodes.

db.local Your local writable feed. You have to get an owner of the hyperdb to authorize you to have your writes replicate. The first person to create the hyperdb is the first owner.

db.authorize(key, [callback]) Authorize another peer to write to the hyperdb.

To get another peer to authorize you you'd usually do something like

```
1 myDb.on('ready', function () {
2   console.log('You local key is ' + myDb.local.key.toString('hex'))
3   console.log('Tell an owner to authorize it')
4 })
```

db.authorized(key, [callback]) Check whether a key is authorized to write to the database.

```
1 myDb.authorized(otherDb.local.key, function (err, auth) {
2   if (err) console.log('err', err)
3   else if (auth === true) console.log('authorized')
4   else console.log('not authorized')
5 })
```

watcher = db.watch(folderOrKey, onchange) Watch a folder and get notified anytime a key inside this folder has changed.

```
1 db.watch('foo/bar', function () {
2   console.log('folder has changed')
3 })
4
5 ...
6
7 db.put('foo/bar/baz', 'hi') // triggers the above
```

You can destroy the watcher by calling `watcher.destroy()`.

The watcher will emit `watching` when it starts watching and `change` when a change has been detected.

If a critical error occurs an error will be emitted on the watcher.

`var stream = db.createReadStream(prefix[, options])` Create a readable stream of nodes stored in the database. Set `prefix` to only iterate nodes prefixed with that folder.

Options include:

```
1 {
2   recursive: true // visit all subfolders.
3               // set to false to only visit the first node in each
4               // folder
5   reverse: true  // read the records in reverse order.
6   gt: false      // visit only strictly nodes that are > than the
   prefix
}
```

`var stream = db.createWriteStream()` Create a writable stream.

Where `stream.write(data)` accepts data as an object or an array of objects with the same form as `db.batch()`.

`db.list(prefix[, options], callback)` Same as `createReadStream` but buffers the result to a list that is passed to the callback.

`var stream = db.createDiffStream(prefix[, checkout])` Find out about changes in key/value pairs between the version `checkout` and current version prefixed by `prefix`.

`stream` is a readable object stream that outputs modifications like

```
1 { left: nodes, right: nodes }
```

`left` are the nodes for a key found in the `db` and `right` are the nodes found in the `checkout`. If no nodes exist in the `db` for the key `left` will be `null` and vice versa.

`var stream = db.createHistoryStream([options])` Returns a readable stream of node objects covering all historic values since the beginning of time.

Nodes are emitted in topographic order, meaning if value `v2` was aware of value `v1` at its insertion time, `v1` must be emitted before `v2`.

To emit the nodes in reverse order pass `{reverse: true}` as an option.

var stream = db.createKeyHistoryStream(key) Returns a readable stream of node objects covering all historic values for a specific key.

Results are returned with the latest value first.

var stream = db.replicate([options]) Create a replication stream. Options include:

```
1 {  
2   live: false // set to true to keep replicating  
3 }
```

License

MIT