lmdb-js Quick Reference

Abbreviated documentation for https://www.npmjs.com/package/lmdb, for easier reference once you already understand how the library works.

```
"use strict";

const lmdb = require("lmdb");

// recommended write strategy: conditional writes

// if path contains . it's a file, otherwise it's a directory, or if null it's in-memory
let db = lmdb.open({ path: "/path/to/db", ... options, ... rootOptions }); // root database
let subDB = db.openDB("sub-db name", ... options);

let options = {
  compression: boolean || { threshold: integer, dictionary: Buffer },
  useVersions: boolean, // entries have version numbers
  sharedStructuresKey: Symbol, // stores values more efficiently by having a central key/structure mapping
  encoding: "msgpack" (default) || "json" || "cbor" || "string" || "ordered-binary" || "binary",
  encoder: object_settings || msgpack: { structuredClone, useFloat32 } || { encode: Function, decode: Function },
  cache: boolean || object_weakLRUCacheSettings, // if enabled, child transactions and rollbacks will not be available
  keyEncoding: "uint32" || "binary" || "ordered-binary" (default),
  keyEncoder: Function,
  dupSort: boolean, // keys have multiple values; use encoding=ordered-binary and getValues(), ifVersion will not be available
  strictAsyncOrder: boolean
};

let rootOptions = {
  path: string,
  maxDbs: integer, // default: 12
  maxReaders: integer,
  overlappingSync: boolean,
```

```
  separateFlushed: boolean,
  pageSize: integer, // set 4096 (default) for fits-in-memory, 8192 for larger especially for range queries
  eventTurnBatching: boolean, // default: true
  txnStartThreshold: integer, // only relevant when eventTurnBatching=false
  encryptionKey: Buffer || string, // 32 bytes
  commitDelay: integer, // in ms
};

// Existence (always synchronous)
let exists = db.doesExist(key);
let exists = db.doesExist(key, version); // for single-value
let exists = db.doesExist(key, value); // for multi-value

// Reads (always synchronous)
let value = db.get(key, options); // value = undefined || single value
let entry = db.getEntry(key, options); // for single-value; entry = undefined || { value, version }
let iterator = db.getValues(key); // for multi-value; iterator<value> (see range/search for special forms)
let version = db.getLastVersion(); // version = integer; version of last `get` call. not available when `cache` is enabled

// Specialized reads
let values = await db.getMany(keys); // optimized db.get that prefetches first to not block main thread
let valueEncoded = db.getBinary(key); // skip value decode

// Range/search (always synchronous)
let iterator = db.getRange(rangeOptions); // iterator<{ key, value }>, has lazy and optionally async map/filter/forEach
let iterator = db.getKeys(rangeOptions); // iterator<value>; key only returned once for multi-value entries
let iterator = db.getValues(key, rangeOptions); // for multi-value; returns all values for a key (start/end affect values, not keys)

let rangeOptions = {
  start: value,
  end: value,
  reverse: boolean,
```

```
offset: integer,
limit: integer,
asArray: boolean // greedy!
};

// Mutations
let success = await db.put(key, value, version, ifVersion); // success = true if stored, false on ifVersion mismatch
let success = await db.remove(key, ifVersion); // for single-value; success = true if deleted, false on ifVersion mismatch
let success = await db.remove(key, value); // for multi-value; success = true if value deleted

// Conditionals
let success = await db.ifVersion(key, ifVersion, () => { ... });
let success = await db.ifNoExists(key, () => { ... });

// Synchronous versions of mutations
let success = db.putSync(key, value, options); // SLOW; versionOrOptions = { append, appendDup, noOverwrite, noDupData, version }
let success = db.removeSync(key, value); // SLOW
let success = db.removeSync(key, ifVersion); // SLOW

// Database-wide mutations
await db.clearAsync(); // removes all entries
db.clearSync(); // same but synchronous
await db.dropAsync(); // remove all entries *and* deletes the database
db.dropSync(); // same but synchronous

// Utilities
lmdb.asBinary(buffer); // Mark buffer as 'already encoded' (stored as-is) rather than literal buffer (which gets type-tagged)
await db.committed; // Wait for all currently pending writes to be committed to the database (in memory)
await db.flushed; // Wait for all currently pending writes to be flushed to disk
await db.prefetch(keys); // Preload specified keys into memory
await db.backup(path); // Stores an internally consistent copy of the database at the specified path
db.on("beforecommit", () => { /* ... */ }); // Fires just before commit to disk, allows async ops, forces eventTurnBatching on
```

```
// Transactions
// A transaction callback is called at some later time when processing database operations, and a single database commit may
//  also contain operations from outside of the transaction, in addition to the transaction. The return value gets passed through.
let returnValue = await db.transaction(() => { /* ... series of DB operations here ... */ });

// Child transactions can be used to support rollbacks; if something throws within one, the changes are rolled back and the
//  transaction will be aborted. Because transaction management is synchronously stateful, no reference to the parent transaction
//  is needed. It's also possible to create a child transaction *outside of* any parent transaction, in which case it will be rolled
//  into a default(?) transaction on the next commit.
let returnValue = await db.childTransaction(() => { /* ... series of DB operations here ... */ });

// Asynchronous transaction callbacks are *possible* but since transaction management is stateful, this can result in unrelated
//  operations unexpectedly ending up in a transaction that they don't belong to. So normally you *shouldn't* await inside of a
//  transaction, even though you would use the asynchronous versions of operations. Instead of checking for failure asynchronously,
//  rely on the transaction abort/rollback.

// There is a synchronous transaction equivalent, which doesn't batch? and executes immediately
let returnValue = db.transactionSync(() => { /* ... series of DB operations here ... */ });

// And there are also read transactions, providing a consistent view to read from
let transaction = db.useReadTransaction();
/* ... read-only operations go here ... */
transaction.done(); // Do not forget! Or it will leak
```

Revision #1
Created 2025-08-22 18:15:04 UTC by joepie91
Updated 2025-08-22 18:16:01 UTC by joepie91