Connection

- Connection.prototype.watch()

Connection()

Parameters

- base «Mongoose» a mongoose instance

Inherits:

- «NodeJS EventEmitter http://nodejs.org/api/events.html#events_class_events_eventemitter»

Connection constructor

For practical reasons, a Connection equals a Db.

Connection.prototype.asPromise()

Returns:

- «Promise»

Returns a promise that resolves when this connection successfully connects to MongoDB, or rejects if this connection failed to connect.

Example:

```
const conn = await mongoose.createConnection('mongodb://localhost:27017/test').
  asPromise();
conn.readyState; // 1, means Mongoose is connected
```

Connection.prototype.client

Type:

- «property»

The MongoClient instance this connection uses to talk to MongoDB. Mongoose automatically sets this property when the connection is opened.

Connection.prototype.close()

Parameters

- [force] «Boolean» optional
- [callback] «Function» optional

Returns:

- «Promise»

Closes the connection

Connection.prototype.collection()

Parameters

- name «String» of the collection
- [options] «Object» optional collection options

Returns:

- «Collection» collection instance

Retrieves a collection, creating it if not cached.

Not typically needed by applications. Just talk to your collection through your model.

Connection.prototype.collections

Type:

- «property»

A hash of the collections associated with this connection

Connection.prototype.config

Type:

- «property»

A hash of the global options that are associated with this connection

---

Connection.prototype.createCollection()

Parameters

- collection «string» The collection to create
- [options] «Object» see MongoDB driver docs
- [callback] «Function»

Returns:

- «Promise»

Helper for `createCollection()`. Will explicitly create the given collection with specified options. Used to create capped collections and views from mongoose.

Options are passed down without modification to the MongoDB driver's `createCollection()` function

---

Connection.prototype.db

Type:

- «property»

The mongodb.Db instance, set when the connection is opened

Connection.prototype.deleteModel()

Parameters

- name «String|RegExp» if string, the name of the model to remove. If regexp, removes all models whose name matches the regexp.

Returns:

- «Connection» this

Removes the model named `name` from this connection, if it exists. You can use this function to clean up any models you created in your tests to prevent OverwriteModelErrors.

Example:

```
conn.model('User', new Schema({ name: String }));
console.log(conn.model('User')); // Model object
conn.deleteModel('User');
console.log(conn.model('User')); // undefined

// Usually useful in a Mocha `afterEach()` hook
afterEach(function() {
  conn.deleteModel(/.+/); // Delete every model
});
```

Connection.prototype.dropCollection()

Parameters

- collection «string» The collection to delete
- [callback] «Function»

Returns:

- «Promise»

Helper for `dropCollection()`. Will delete the given collection, including all documents and indexes.

Connection.prototype.dropDatabase()

Parameters

- [callback] «Function»

Returns:

- «Promise»

Helper for `dropDatabase()`. Deletes the given database, including all collections, documents, and indexes.

Example:

```
const conn = mongoose.createConnection('mongodb://localhost:27017/mydb');
// Deletes the entire 'mydb' database
await conn.dropDatabase();
```

Connection.prototype.get()

Parameters

- key «String»

Gets the value of the option `key`. Equivalent to `conn.options[key]`

Example:

```
conn.get('test'); // returns the 'test' value
```

Connection.prototype.getClient()

Returns:

- «MongoClient»

Returns the MongoDB driver `MongoClient` instance that this connection uses to talk to MongoDB.

Example:

```
const conn = await mongoose.createConnection('mongodb://localhost:27017/test');

conn.getClient(); // MongoClient { ... }
```

Connection.prototype.host

Type:

- «property»

The host name portion of the URI. If multiple hosts, such as a replica set, this will contain the first host name in the URI

Example

```
mongoose.createConnection('mongodb://localhost:27017/mydb').host; // "localhost"
```

Connection.prototype.id

Type:

- «property»

A number identifier for this connection. Used for debugging when you have multiple connections.

Example

```
// The default connection has `id = 0`
mongoose.connection.id; // 0

// If you create a new connection, Mongoose increments id
const conn = mongoose.createConnection();
conn.id; // 1
```

Connection.prototype.model()

Parameters

- name «String|Function» the model name or class extending Model
- [schema] «Schema» a schema. necessary when defining a model
- [collection] «String» name of mongodb collection (optional) if not given it will be induced from model name
- [options] «Object»
  - [options.overwriteModels=false] «Boolean» If true, overwrite existing models with the same name to avoid `OverwriteModelError`

Returns:

- «Model» The compiled model

Defines or retrieves a model.

```
const mongoose = require('mongoose');
const db = mongoose.createConnection(..);
db.model('Venue', new Schema(..));
const Ticket = db.model('Ticket', new Schema(..));
const Venue = db.model('Venue');
```

When no `collection` argument is passed, Mongoose produces a collection name by passing the model `name` to the utils.toCollectionName method. This method pluralizes the name. If you don't like this behavior, either pass a collection name or set your schemas collection name option.

Example:

```
const schema = new Schema({ name: String }, { collection: 'actor' });

// or

schema.set('collection', 'actor');

// or

const collectionName = 'actor'
const M = conn.model('Actor', schema, collectionName)
```

Connection.prototype.modelNames()

Returns:

- «Array»

Returns an array of model names created on this connection.

Connection.prototype.models

Type:

- «property»

A POJO containing a map from model names to models. Contains all models that have been added to this connection using `Connection#model()`.

Example

```
const conn = mongoose.createConnection();
const Test = conn.model('Test', mongoose.Schema({ name: String }));

Object.keys(conn.models).length; // 1
conn.models.Test === Test; // true
```

Connection.prototype.name

Type:

- «property»

The name of the database this connection points to.

Example

```
mongoose.createConnection('mongodb://localhost:27017/mydb').name; // "mydb"
```

Connection.prototype.openUri()

Parameters

- uri «String» The URI to connect with.
- [options] «Object» Passed on to http://mongodb.github.io/node-mongodb-native/2.2/api/MongoClient.html#connect
    - [options.bufferCommands=true] «Boolean» Mongoose specific option. Set to false to disable buffering on all models associated with this connection.
    - [options.bufferTimeoutMS=10000] «Number» Mongoose specific option. If `bufferCommands` is true, Mongoose will throw an error after `bufferTimeoutMS` if the operation is still buffered.

- [options.dbName] «String» The name of the database we want to use. If not provided, use database name from connection string.
- [options.user] «String» username for authentication, equivalent to `options.auth.user`. Maintained for backwards compatibility.
- [options.pass] «String» password for authentication, equivalent to `options.auth.password`. Maintained for backwards compatibility.
- [options.maxPoolSize=100] «Number» The maximum number of sockets the MongoDB driver will keep open for this connection. Keep in mind that MongoDB only allows one operation per socket at a time, so you may want to increase this if you find you have a few slow queries that are blocking faster queries from proceeding. See Slow Trains in MongoDB and Node.js.
- [options.minPoolSize=0] «Number» The minimum number of sockets the MongoDB driver will keep open for this connection. Keep in mind that MongoDB only allows one operation per socket at a time, so you may want to increase this if you find you have a few slow queries that are blocking faster queries from proceeding. See Slow Trains in MongoDB and Node.js.
- [options.serverSelectionTimeoutMS] «Number» If `useUnifiedTopology = true`, the MongoDB driver will try to find a server to send any given operation to, and keep retrying for `serverSelectionTimeoutMS` milliseconds before erroring out. If not set, the MongoDB driver defaults to using `30000` (30 seconds).
- [options.heartbeatFrequencyMS] «Number» If `useUnifiedTopology = true`, the MongoDB driver sends a heartbeat every `heartbeatFrequencyMS` to check on the status of the connection. A heartbeat is subject to `serverSelectionTimeoutMS`, so the MongoDB driver will retry failed heartbeats for up to 30 seconds by default. Mongoose only emits a `'disconnected'` event after a heartbeat has failed, so you may want to decrease this setting to reduce the time between when your server goes down and when Mongoose emits `'disconnected'`. We recommend you do **not** set this setting below 1000, too many heartbeats can lead to performance degradation.
- [options.autoIndex=true] «Boolean» Mongoose-specific option. Set to false to disable automatic index creation for all models associated with this connection.
- [options.promiseLibrary] «Class» Sets the underlying driver's promise library.
- [options.connectTimeoutMS=30000] «Number» How long the MongoDB driver will wait before killing a socket due to inactivity *during initial connection*. Defaults to 30000. This option is passed transparently to Node.js' `socket#setTimeout()` function.
- [options.socketTimeoutMS=30000] «Number» How long the MongoDB driver will wait before killing a socket due to inactivity *after initial connection*. A socket may be inactive because of either no activity or a long-running operation. This is set to `30000` by default, you should set this to 2-3x your longest running operation if you expect some of your database operations to run longer than 20 seconds. This option is passed to Node.js `socket#setTimeout()` function after the MongoDB driver successfully completes.
- [options.family=0] «Number» Passed transparently to Node.js' `dns.lookup()` function. May be either `0`, `4`, `or` `6`. `4` means use IPv4 only, `6` means use IPv6 only, `0` means try both.

    - [options.autoCreate=false] «Boolean» Set to `true` to make Mongoose automatically call `createCollection()` on every model created on this connection.

- [callback] «Function»

Opens the connection with a URI using `MongoClient.connect()`.

Connection.prototype.pass

Type:

- «property»

The password specified in the URI

Example

```
mongoose.createConnection('mongodb://val:psw@localhost:27017/mydb').pass; // "psw"
```

Connection.prototype.plugin()

Parameters

- fn «Function» plugin callback
- [opts] «Object» optional options

Returns:

- «Connection» this

Declares a plugin executed on all schemas you pass to `conn.model()`

Equivalent to calling `.plugin(fn)` on each schema you create.

Example:

```
const db = mongoose.createConnection('mongodb://localhost:27017/mydb');
db.plugin(() => console.log('Applied'));
db.plugins.length; // 1

db.model('Test', new Schema({})); // Prints "Applied"
```

Connection.prototype.plugins

Type:

- «property»

The plugins that will be applied to all models created on this connection.

Example:

```
const db = mongoose.createConnection('mongodb://localhost:27017/mydb');
db.plugin(() => console.log('Applied'));
db.plugins.length; // 1

db.model('Test', new Schema({})); // Prints "Applied"
```

Connection.prototype.port

Type:

- «property»

The port portion of the URI. If multiple hosts, such as a replica set, this will contain the port from the first host name in the URI.

Example

```
mongoose.createConnection('mongodb://localhost:27017/mydb').port; // 27017
```

Connection.prototype.readyState

Type:

- «property»

Connection ready state

- 0 = disconnected
- 1 = connected
- 2 = connecting
- 3 = disconnecting

Each state change emits its associated event name.

Example

```
conn.on('connected', callback);
conn.on('disconnected', callback);
```

Connection.prototype.set()

Parameters

- key «String»
- val «Any»

Sets the value of the option `key`. Equivalent to `conn.options[key] = val`

Supported options include

- `maxTimeMS`: Set `maxTimeMS` for all queries on this connection.

Example:

```
conn.set('test', 'foo');
conn.get('test'); // 'foo'
conn.options.test; // 'foo'
```

Connection.prototype.setClient()

Returns:

- «Connection» this

Set the MongoDB driver `MongoClient` instance that this connection uses to talk to MongoDB. This is useful if you already have a MongoClient instance, and want to reuse it.

Example:

```
const client = await mongodb.MongoClient.connect('mongodb://localhost:27017/test');

const conn = mongoose.createConnection().setClient(client);

conn.getClient(); // MongoClient { ... }
conn.readyState; // 1, means 'CONNECTED'
```

Connection.prototype.startSession()

Parameters

- [options] «Object» see the mongodb driver options
    - [options.causalConsistency=true] «Boolean» set to false to disable causal consistency

- [callback] «Function»

Returns:

- «Promise<ClientSession>» promise that resolves to a MongoDB driver `ClientSession`

Requires MongoDB >= 3.6.0. Starts a MongoDB session for benefits like causal consistency, retryable writes, and transactions.

Example:

```
const session = await conn.startSession();
let doc = await Person.findOne({ name: 'Ned Stark' }, null, { session });
await doc.remove();
// `doc` will always be null, even if reading from a replica set
// secondary. Without causal consistency, it is possible to
// get a doc back from the below query if the query reads from a
// secondary that is experiencing replication lag.
doc = await Person.findOne({ name: 'Ned Stark' }, null, { session, readPreference: 'secondary' });
```

Connection.prototype.transaction()

Parameters

- fn «Function» Function to execute in a transaction
- [options] «mongodb.TransactionOptions» Optional settings for the transaction

Returns:

- «Promise<Any>» promise that is fulfilled if Mongoose successfully committed the transaction, or rejects if the transaction was aborted or if Mongoose failed to commit the transaction. If fulfilled, the promise resolves to a MongoDB command result.

Requires MongoDB >= 3.6.0. Executes the wrapped async function in a transaction. Mongoose will commit the transaction if the async function executes successfully and attempt to retry if there was a retriable error.

Calls the MongoDB driver's `session.withTransaction()`, but also handles resetting Mongoose document state as shown below.

Example:

```
const doc = new Person({ name: 'Will Riker' });
await db.transaction(async function setRank(session) {
  doc.rank = 'Captain';
  await doc.save({ session });
  doc.isNew; // false

  // Throw an error to abort the transaction
  throw new Error('Oops!');
```

```
},{ readPreference: 'primary' }).catch(() => {});

// true, `transaction()` reset the document's state because the
// transaction was aborted.
doc.isNew;
```

Connection.prototype.useDb()

Parameters

- name «String» The database name
- [options] «Object»
  - [options.useCache=false] «Boolean» If true, cache results so calling `useDb()` multiple times with the same name only creates 1 connection object.
  - [options.noListener=false] «Boolean» If true, the connection object will not make the db listen to events on the original connection. See issue #9961.

Returns:

- «Connection» New Connection Object

Switches to a different database using the same connection pool.

Returns a new connection object, with the new db.

Connection.prototype.user

Type:

- «property»

The username specified in the URI

Example

```
mongoose.createConnection('mongodb://val:psw@localhost:27017/mydb').user; // "val"
```

Connection.prototype.watch()

Parameters

- [pipeline] «Array»

- [options] «Object» passed without changes to the MongoDB driver's `Db#watch()` function

Returns:

- «ChangeStream» mongoose-specific change stream wrapper, inherits from EventEmitter

Watches the entire underlying database for changes. Similar to `Model.watch()`.

This function does **not** trigger any middleware. In particular, it does **not** trigger aggregate middleware.

The ChangeStream object is an event emitter that emits the following events

- 'change': A change occurred, see below example
- 'error': An unrecoverable error occurred. In particular, change streams currently error out if they lose connection to the replica set primary. Follow this GitHub issue for updates.
- 'end': Emitted if the underlying stream is closed
- 'close': Emitted if the underlying stream is closed

Example:

```
const User = conn.model('User', new Schema({ name: String }));

const changeStream = conn.watch().on('change', data => console.log(data));

// Triggers a 'change' event on the change stream.
await User.create({ name: 'test' });
```