Connections

You can connect to MongoDB with the `mongoose.connect()` method.

```
mongoose.connect('mongodb://localhost:27017/myapp');
```

This is the minimum needed to connect the `myapp` database running locally on the default port (27017). If connecting fails on your machine, try using `127.0.0.1` instead of `localhost`.

You can also specify several more parameters in the `uri`:

```
mongoose.connect('mongodb://username:password@host:port/database?options...');
```

See the mongodb connection string spec for more details.

Operation Buffering

Mongoose lets you start using your models immediately, without waiting for mongoose to establish a connection to MongoDB.

```
mongoose.connect('mongodb://localhost:27017/myapp');
const MyModel = mongoose.model('Test', new Schema({ name: String }));
// Works
MyModel.findOne(function(error, result) { /* ... */ });
```

That's because mongoose buffers model function calls internally. This buffering is convenient, but also a common source of confusion. Mongoose will *not* throw any errors by default if you use a model without connecting.

```
const MyModel = mongoose.model('Test', new Schema({ name: String }));
// Will just hang until mongoose successfully connects
MyModel.findOne(function(error, result) { /* ... */ });
```

```
setTimeout(function() {
  mongoose.connect('mongodb://localhost:27017/myapp');
}, 60000);
```

To disable buffering, turn off the `bufferCommands` option on your schema. If you have `bufferCommands` on and your connection is hanging, try turning `bufferCommands` off to see if you haven't opened a connection properly. You can also disable `bufferCommands` globally:

```
mongoose.set('bufferCommands', false);
```

Note that buffering is also responsible for waiting until Mongoose creates collections if you use the `autoCreate` option. If you disable buffering, you should also disable the `autoCreate` option and use `createCollection()` to create capped collections or collections with collations.

```
const schema = new Schema({
  name: String
}, {
  capped: { size: 1024 },
  bufferCommands: false,
  autoCreate: false // disable `autoCreate` since `bufferCommands` is false
});

const Model = mongoose.model('Test', schema);
// Explicitly create the collection before using it
// so the collection is capped.
await Model.createCollection();
```

Error Handling

There are two classes of errors that can occur with a Mongoose connection.

- Error on initial connection. If initial connection fails, Mongoose will emit an 'error' event and the promise `mongoose.connect()` returns will reject. However, Mongoose will **not** automatically try to reconnect.
- Error after initial connection was established. Mongoose will attempt to reconnect, and it will emit an 'error' event.

To handle initial connection errors, you should use `.catch()` or `try/catch` with async/await.

```
mongoose.connect('mongodb://localhost:27017/test').
  catch(error => handleError(error));

// Or:
try {
  await mongoose.connect('mongodb://localhost:27017/test');
} catch (error) {
  handleError(error);
}
```

To handle errors after initial connection was established, you should listen for error events on the connection. However, you still need to handle initial connection errors as shown above.

```
mongoose.connection.on('error', err => {
  logError(err);
});
```

Note that Mongoose does not necessarily emit an 'error' event if it loses connectivity to MongoDB. You should listen to the `disconnected` event to report when Mongoose is disconnected from MongoDB.

Options

The `connect` method also accepts an `options` object which will be passed on to the underlying MongoDB driver.

```
mongoose.connect(uri, options);
```

A full list of options can be found on the MongoDB Node.js driver docs for `connect()`. Mongoose passes options to the driver without modification, modulo a few exceptions that are explained below.

- `bufferCommands` - This is a mongoose-specific option (not passed to the MongoDB driver) that disables Mongoose's buffering mechanism
- `user` / `pass` - The username and password for authentication. These options are Mongoose-specific, they are equivalent to the MongoDB driver's `auth.username` and `auth.password` options.
- `autoIndex` - By default, mongoose will automatically build indexes defined in your schema when it connects. This is great for development, but not ideal for large production deployments, because index builds can cause performance degradation. If you set `autoIndex` to false, mongoose will not automatically build indexes for **any** model associated with this connection.
- `dbName` - Specifies which database to connect to and overrides any database specified in the connection string. This is useful if you are unable to specify a default database in the connection string like with some `mongodb+srv` syntax connections.

Below are some of the options that are important for tuning Mongoose.

- `promiseLibrary` - Sets the underlying driver's promise library.
- `maxPoolSize` - The maximum number of sockets the MongoDB driver will keep open for this connection. By default, `maxPoolSize` is 100. 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. You may want to decrease `maxPoolSize` if you are running into connection limits.
- `minPoolSize` - The minimum number of sockets the MongoDB driver will keep open for this connection. The MongoDB driver may close sockets that have been inactive for some time. You may want to increase `minPoolSize` if you expect your app to go through long idle times and want to make sure your sockets stay open to avoid slow trains when activity picks up.
- `socketTimeoutMS` - 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.
- `family` - Whether to connect using IPv4 or IPv6. This option passed to Node.js' `dns.lookup()` function. If you don't specify this option, the MongoDB driver will try IPv6 first and then IPv4 if IPv6 fails. If your `mongoose.connect(uri)` call takes a long time, try `mongoose.connect(uri, { family: 4 })`
- `authSource` - The database to use when authenticating with `user` and `pass`. In MongoDB, users are scoped to a database. If you are getting an unexpected login failure, you may need to set this option.
- `serverSelectionTimeoutMS` - The MongoDB driver will try to find a server to send any given operation to, and keep retrying for `serverSelectionTimeoutMS` milliseconds. If not set, the MongoDB driver defaults to using `30000` (30 seconds).
- `heartbeatFrequencyMS` - 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.

The `serverSelectionTimeoutMS` option also handles how long `mongoose.connect()` will retry initial connection before erroring out. `mongoose.connect()` will retry for 30 seconds by default (default `serverSelectionTimeoutMS`) before erroring out. To get faster feedback on failed operations, you can reduce `serverSelectionTimeoutMS` to 5000 as shown below.

Example:

```
const options = {
  autoIndex: false, // Don't build indexes
  maxPoolSize: 10, // Maintain up to 10 socket connections
  serverSelectionTimeoutMS: 5000, // Keep trying to send operations for 5 seconds
  socketTimeoutMS: 45000, // Close sockets after 45 seconds of inactivity
  family: 4 // Use IPv4, skip trying IPv6
};
mongoose.connect(uri, options);
```

See this page for more information about `connectTimeoutMS` and `socketTimeoutMS`

Callback

The `connect()` function also accepts a callback parameter and returns a promise.

```
mongoose.connect(uri, options, function(error) {
  // Check error in initial connection. There is no 2nd param to the callback.
});

// Or using promises
mongoose.connect(uri, options).then(
  () => { /** ready to use. The `mongoose.connect()` promise resolves to mongoose instance. */ },
  err => { /** handle initial connection error */ }
);
```

Connection String Options

You can also specify driver options in your connection string as parameters in the query string portion of the URI. This only applies to options passed to the MongoDB driver. You **can't** set Mongoose-specific options like `bufferCommands` in the query string.

```
mongoose.connect('mongodb://localhost:27017/test?connectTimeoutMS=1000&bufferCommands=false&authSc
// The above is equivalent to:
mongoose.connect('mongodb://localhost:27017/test', {
  connectTimeoutMS: 1000
  // Note that mongoose will **not** pull `bufferCommands` from the query string
});
```

The disadvantage of putting options in the query string is that query string options are harder to read. The advantage is that you only need a single configuration option, the URI, rather than separate options for `socketTimeoutMS`, `connectTimeoutMS`, etc. Best practice is to put options that likely differ between development and production, like `replicaSet` or `ssl`, in the connection string, and options that should remain constant, like `connectTimeoutMS` or `maxPoolSize`, in the options object.

The MongoDB docs have a full list of supported connection string options. Below are some options that are often useful to set in the connection string because they are closely associated with the hostname and authentication information.

- `authSource` - The database to use when authenticating with `user` and `pass`. In MongoDB, users are scoped to a database. If you are getting an unexpected login failure, you may need to set this option.
- `family` - Whether to connect using IPv4 or IPv6. This option passed to Node.js' `dns.lookup()` function. If you don't specify this option, the MongoDB driver will try IPv6 first and then IPv4 if IPv6 fails. If your `mongoose.connect(uri)` call takes a long time, try `mongoose.connect(uri, { family: 4 })`

Connection Events

Connections inherit from Node.js' `EventEmitter` class, and emit events when something happens to the connection, like losing connectivity to the MongoDB server. Below is a list of events that a connection may emit.

- `connecting`: Emitted when Mongoose starts making its initial connection to the MongoDB server
- `connected`: Emitted when Mongoose successfully makes its initial connection to the MongoDB server, or when Mongoose reconnects after losing connectivity.
- `open`: Equivalent to `connected`
- `disconnecting`: Your app called `Connection#close()` to disconnect from MongoDB
- `disconnected`: Emitted when Mongoose lost connection to the MongoDB server. This event may be due to your code explicitly closing the connection, the database server crashing, or network connectivity issues.
- `close`: Emitted after `Connection#close()` successfully closes the connection. If you call `conn.close()`, you'll get both a 'disconnected' event and a 'close' event.
- `reconnected`: Emitted if Mongoose lost connectivity to MongoDB and successfully reconnected. Mongoose attempts to automatically reconnect when it loses connection to the database.

- `error`: Emitted if an error occurs on a connection, like a `parseError` due to malformed data or a payload larger than 16MB.
- `fullsetup`: Emitted when you're connecting to a replica set and Mongoose has successfully connected to the primary and at least one secondary.
- `all`: Emitted when you're connecting to a replica set and Mongoose has successfully connected to all servers specified in your connection string.
- `reconnectFailed`: Emitted when you're connected to a standalone server and Mongoose has run out of `reconnectTries`. The MongoDB driver will no longer attempt to reconnect after this event is emitted. This event will never be emitted if you're connected to a replica set.

When you're connecting to a single MongoDB server (a "standalone"), Mongoose will emit 'disconnected' if it gets disconnected from the standalone server, and 'connected' if it successfully connects to the standalone. In a replica set, Mongoose will emit 'disconnected' if it loses connectivity to the replica set primary, and 'connected' if it manages to reconnect to a the replica set primary.

A note about keepAlive

For long running applications, it is often prudent to enable `keepAlive` with a number of milliseconds. Without it, after some period of time you may start to see `"connection closed"` errors for what seems like no reason. If so, after reading this, you may decide to enable `keepAlive`:

```
mongoose.connect(uri, { keepAlive: true, keepAliveInitialDelay: 300000 });
```

`keepAliveInitialDelay` is the number of milliseconds to wait before initiating `keepAlive` on the socket. `keepAlive` is true by default since mongoose 5.2.0.

Replica Set Connections

To connect to a replica set you pass a comma delimited list of hosts to connect to rather than a single host.

```
mongoose.connect('mongodb://[username:password@]host1[:port1][,host2[:port2],...[,hostN[:portN]]][
```

For example:

```
mongoose.connect('mongodb://user:pw@host1.com:27017,host2.com:27017,host3.com:27017/testdb');
```

To connect to a single node replica set, specify the `replicaSet` option.

```
mongoose.connect('mongodb://host1:port1/?replicaSet=rsName');
```

Server Selection

The underlying MongoDB driver uses a process known as server selection to connect to MongoDB and send operations to MongoDB. If the MongoDB driver can't find a server to send an operation to after `serverSelectionTimeoutMS`, you'll get the below error:

```
MongoTimeoutError: Server selection timed out after 30000 ms
```

You can configure the timeout using the `serverSelectionTimeoutMS` option to `mongoose.connect()`:

```
mongoose.connect(uri, {
  serverSelectionTimeoutMS: 5000 // Timeout after 5s instead of 30s
});
```

A `MongoTimeoutError` has a `reason` property that explains why server selection timed out. For example, if you're connecting to a standalone server with an incorrect password, `reason` will contain an "Authentication failed" error.

```
const mongoose = require('mongoose');

const uri = 'mongodb+srv://username:badpw@cluster0-OMITTED.mongodb.net/' +
  'test?retryWrites=true&w=majority';
// Prints "MongoServerError: bad auth Authentication failed."
mongoose.connect(uri, {
  serverSelectionTimeoutMS: 5000
}).catch(err => console.log(err.reason));
```

Replica Set Host Names

MongoDB replica sets rely on being able to reliably figure out the domain name for each member. On Linux and OSX, the MongoDB server uses the output of the `hostname` command to figure out the domain name to report to the replica set. This can cause confusing errors if you're connecting to a remote MongoDB replica set running on a machine that reports its `hostname` as `localhost`:

```
// Can get this error even if your connection string doesn't include
// `localhost` if `rs.conf()` reports that one replica set member has
// `localhost` as its host name.
failed to connect to server [localhost:27017] on first connect
```

If you're experiencing a similar error, connect to the replica set using the `mongo` shell and run the `rs.conf()` command to check the host names of each replica set member. Follow this page's instructions to change a replica set member's host name.

Multi-mongos support

You can also connect to multiple mongos instances for high availability in a sharded cluster. You do not need to pass any special options to connect to multiple mongos in mongoose 5.x.

```
// Connect to 2 mongos servers
mongoose.connect('mongodb://mongosA:27501,mongosB:27501', cb);
```

Multiple connections

So far we've seen how to connect to MongoDB using Mongoose's default connection. Mongoose creates a *default connection* when you call `mongoose.connect()`. You can access the default connection using

`mongoose.connection`.

You may need multiple connections to MongoDB for several reasons. One reason is if you have multiple databases or multiple MongoDB clusters. Another reason is to work around slow trains. The `mongoose.createConnection()` function takes the same arguments as `mongoose.connect()` and returns a new connection.

```
const conn = mongoose.createConnection('mongodb://[username:password@]host1[:port1][,host2[:port2]
```

This connection object is then used to create and retrieve models. Models are **always** scoped to a single connection.

```
const UserModel = conn.model('User', userSchema);
```

If you use multiple connections, you should make sure you export schemas, **not** models. Exporting a model from a file is called the *export model pattern*. The export model pattern is limited because you can only use one connection.

```
const userSchema = new Schema({ name: String, email: String });

// The alternative to the export model pattern is the export schema pattern.
module.exports = userSchema;

// Because if you export a model as shown below, the model will be scoped
// to Mongoose's default connection.
// module.exports = mongoose.model('User', userSchema);
```

If you use the export schema pattern, you still need to create models somewhere. There are two common patterns. First is to export a connection and register the models on the connection in the file:

```
// connections/fast.js
const mongoose = require('mongoose');

const conn = mongoose.createConnection(process.env.MONGODB_URI);
conn.model('User', require('../schemas/user'));

module.exports = conn;

// connections/slow.js
const mongoose = require('mongoose');

const conn = mongoose.createConnection(process.env.MONGODB_URI);
conn.model('User', require('../schemas/user'));
conn.model('PageView', require('../schemas/pageView'));

module.exports = conn;
```

Another alternative is to register connections with a dependency injector or another inversion of control (IOC) pattern.

```
const mongoose = require('mongoose');

module.exports = function connectionFactory() {
  const conn = mongoose.createConnection(process.env.MONGODB_URI);

  conn.model('User', require('../schemas/user'));
  conn.model('PageView', require('../schemas/pageView'));

  return conn;
};
```

Connection Pools

Each `connection`, whether created with `mongoose.connect` or `mongoose.createConnection` are all backed by an internal configurable connection pool defaulting to a maximum size of 100. Adjust the pool size using your connection options:

```
// With object options
mongoose.createConnection(uri, { maxPoolSize: 10 });

// With connection string options
const uri = 'mongodb://localhost:27017/test?maxPoolSize=10';
mongoose.createConnection(uri);
```

Next Up

Now that we've covered connections, let's take a look at models.