Administration

On this page

- Databases
- Collections
- Authentication
- Logger
- Monitoring

Databases

The driver provides various helpers on database objects for executing commands, getting collection lists, and administrative tasks.

List Collections

To get a list of collections or collection names for a database, use `collections` and `collection_names`, respectively.

```
client = Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'music')
database = client.database

database.collections # Returns an array of Collection objects.
database.collection_names # Returns an array of collection names as strings.
```

To execute any command on the database, use the `command` method.

```
client = Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'music')
database = client.database

result = database.command(:ismaster => 1)
result.first # Returns the BSON::Document returned from the server.
```

Drop Database

To drop a database, use the `drop` method.

```
client = Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'music')
client.database.drop
```

Collections

The driver provides some helpers for administrative tasks with collections.

To create a collection with options (such as creating a capped collection), pass the options when getting the collection from the client, then call `create`.

```
client = Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'music')
artists = client[:artists, :capped => true, :size => 1024]
artists.create
artists.capped? # Returns true.
```

Drop Collection

To drop a collection, call `drop` on the collection object.

```
client = Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'music')
artists = client[:artists]
artists.drop
```

Changing Read/Write Preferences

To change the default read preference or write concern for specific operations, use the `with` method on the collection.

```
client = Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'music')
artists = client[:artists]
artists.with(:read => { :mode => :primary_preferred }).find.to_a
artists.with(:write => { :w => :3 }).insert_one( { :name => 'Depeche Mode' } )
```

Authentication

MongoDB supports a variety of authentication mechanisms.

For more information about configuring your MongoDB server for each of these authentication mechanisms see MongoDB's online documentation.

Creating a user

To create a user in specific database, use the `create` method with the username, password and roles parameters.

```
client.database.users.create(
      'durran',
      password: 'password',
      roles: [ Mongo::Auth::Roles::READ_WRITE ])
```

SEE ALSO:

Built-in roles

Providing credentials

If authentication is enabled, provide credentials when creating a new client.

```
client = Mongo::Client.new([ '127.0.0.1:27017' ],
                           user: 'test',
                           password: '123' )
```

For MongoDB 2.6 and later, `:auth_source` defaults to **admin**, otherwise the current database is used.

The current database can be changed with the client's `use` method.

```
client = Mongo::Client.new([ '127.0.0.1:27017' ])
music_client = client.use( 'music')
```

A new client can be created with the authentication credentials.

```
authenticated_client = client.with( user: 'test',
                                    password: '123' )
```

Alternatively, setting the current database and credentials can be done in one step:

```
authenticated_music_client = client.with( :database => 'music',
                                          user:'test',
                                          password:'123' )
```

MONGODB-CR Mechanism

MONGODB-CR was the default authentication mechanism for MongoDB up through version 2.6.

The mechanism can be explicitly set with the credentials:

```
client = Mongo::Client.new([ '127.0.0.1:27017' ],
                           :database => 'music',
                           user: 'test',
                           password: '123',
                           auth_mech: :mongodb_cr )
```

Client Certificate (x509)

Requires MongoDB v2.6 or greater.

The driver presents an X.509 certificate during SSL negotiation. The Client Certificate (x509) mechanism authenticates a username derived from the distinguished subject name of this certificate.

This authentication method requires the use of SSL connections with certificate validation.

For more information about configuring X.509 authentication in MongoDB, see the X.509 tutorial in the MongoDB Manual.

```
client = Mongo::Client.new([ '127.0.0.1:27017' ],
                           auth_mech: :mongodb_x509,
                           ssl: true,
                           ssl_cert: '/path/to/client.pem',
                           ssl_ca_cert: '/path/to/ca.pem' )
```

LDAP (SASL PLAIN) mechanism

Requires MongoDB Enterprise Edition v2.6 or greater.

MongoDB Enterprise Edition supports the LDAP authentication mechanism which allows you to delegate authentication using a Lightweight Directory Access Protocol LDAP server.

> **WARNING:**
> When using LDAP, passwords are sent to the server in plain text. For this reason, we strongly recommend enabling SSL when using LDAP as your authentication mechanism.

For more information about configuring LDAP authentication in MongoDB, see the SASL/LDAP tutorial in the MongoDB Manual.

```
client = Mongo::Client.new([ '127.0.0.1:27017' ],
                           auth_mech: :plain,
                           ssl: true,
                           ssl_verify: true,
                           ssl_cert: '/path/to/client.pem',
                           ssl_ca_cert: '/path/to/ca.pem' )
```

Kerberos (GSSAPI) mechanism

Requires MongoDB Enterprise Edition v2.4 or greater.

MongoDB Enterprise Edition v2.4+ supports Kerberos authentication.

To use Kerberos in the Ruby driver with **JRuby**, do the following:

1. Specify several system properties so that the underlying GSSAPI Java libraries can acquire a Kerberos ticket. See the MongoDB Java Driver authentication documentation for more information.
2. Either provide a password OR set the 'java.security.auth.login.config' system property to a config file that references a keytab file.

To use Kerberos in the Ruby driver with **Matz's Ruby Interpreter (MRI)**, create a ticket-granting ticket using `kinit`. See this documentation for more information.

For more information about deploying MongoDB with Kerberos authentication, see the manual.

```
client = Mongo::Client.new([ '127.0.0.1:27017' ],
                           auth_mech: :gssapi,
                           user: 'test',
                           password: '123' )
```

Logger

You can either use the default global driver logger or set your own. To set your own:

```
Mongo::Logger.logger = other_logger
```

See the Ruby Logger documentation for more information on the default logger API and available levels.

Changing the Logger Level

To change the logger level:

```
Mongo::Logger.logger.level = Logger::WARN
```

For more control, a logger can be passed to a client for per-client control over logging.

```
my_logger = Logger.new($stdout)
Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'test', :logger => my_logger )
```

Truncation

The default logging truncates logs at 250 characters by default. To turn this off pass an option to the client instance.

```
Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'test', :truncate_logs => false )
```

Monitoring

All user-initiated commands that are sent to the server publish events that can be subscribed to for fine grained information. The monitoring API publishes a guaranteed start event for each command, then either a succeeded or failed event. A subscriber must implement 3 methods: `started`, `succeeded`, and `failed`, each which takes a single parameter for the event. An example is the default logging subscriber included in the driver:

```
module Mongo
  class Monitoring
    class CommandLogSubscriber
      include Loggable

      attr_reader :options

      LOG_STRING_LIMIT = 250

      def initialize(options = {})
        @options = options
      end

      def started(event)
        log_debug("#{prefix(event)} | STARTED | #{format_command(event.command)}")
      end

      def succeeded(event)
        log_debug("#{prefix(event)} | SUCCEEDED | #{event.duration}s")
      end

      def failed(event)
        log_debug("#{prefix(event)} | FAILED | #{event.message} | #{event.duration}s")
      end

      private

      def format_command(args)
        begin
          truncating? ? truncate(args) : args.inspect
        rescue Exception
          '<Unable to inspect arguments>'
        end
      end

      def prefix(event)
        "#{event.address.to_s} | #{event.database_name}.#{event.command_name}"
      end

      def truncate(command)
```

```
        ((s = command.inspect).length > LOG_STRING_LIMIT) ? "#{s[0..LOG_STRING_LIMIT]}..." : s
      end

      def truncating?
        @truncating ||= (options[:truncate_logs] != false)
      end
    end
  end
end
```

To register a custom subscriber, you can do so globally for all clients or on a per-client basis:

```
Mongo::Monitoring::Global.subscribe(Mongo::Monitoring::COMMAND, my_subscriber)

client = Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'test' )
client.subscribe( Mongo::Monitoring::COMMAND, my_subscriber )
```

To turn off monitoring, set the client monitoring option to `false`:

```
client = Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'test', :monitoring => false )
```