MongoDB C# Driver Tutorial

This tutorial is an introduction to the 10gen supported C# driver for MongoDB. It assumes some familiarity with MongoDB and therefore focuses mainly on how to access MongoDB using C#. This tutorial discusses the major classes provided by the C# driver: MongoServer, MongoDatabase, MongoCollection, MongoCursor, MongoGridFS, MongoGridFSFileInfo, and SafeMode, as well as classes from the BSON Library: BsonElement, BsonDocument, BsonArray, and a number of classes used to represent values of various BSON types.

This tutorial is organized in a top down fashion, so you may sometimes have to peek ahead if something isn’t making sense right away.

# References and namespaces

In order to use the C# driver from your program you must add references to the following two DLLs:

* BsonLibrary.dll
* MongoDBClient.dll

You also should add the following using statements to your source files:

using MongoDB.BsonLibrary;

using MongoDB.MongoDBClient

The C# driver is packaged as two DLLs because the BSON layer is usable by itself in contexts outside that of a MongoDB client driver and is therefore packaged as a separate DLL.

With very few exceptions the names of classes that you will be using are prefixed with either “Bson” if they come from the BsonLibrary or “Mongo” if they come from the C# driver. This is done to minimize the chances of getting name collisions when you add the two using statements to your program. Some classes that are expected to be used in method arguments (mainly enums and flags) have shorter names that don’t use either of the prefixes.

We recommend that you use C#’s var statement to declare your variables as it leads to shorter, and we feel, more readable code. Visual Studio makes it easy to see a variable’s type by either hovering the mouse pointer over the variable or by using Intellisense. However, when reading this document you don’t have that capability, so in this document, rather than writing:

var server = MongoServer.Create(connectionString);

var test = server[“test”];

var books = test.GetCollection<BsonDocument>(“books”);

as we recommend, we will instead write:

MongoServer server = MongoServer.Create(connectionString);

MongoDatabase test = server[“test”];

MongoCollection<BsonDocument> books =

test.GetCollection<BsonDocument>(“books”);

so that you can see exactly what types are being used.

# MongoServer class

This class serves as the root object for working with MongoDB. An instance of this class on the client represents a MongoDB server that you wish to communicate with. While this class does have public constructors, the recommended way to get an instance of this class is to use the Create factory method.

Each instance of MongoServer maintains a pool of connections to the server. These connections are shared among all the calls to the server. One of the few reasons you might want to occassionally call the constructor for MongoServer directly instead of calling the Create factory method is if you want to maintain a separate connection pool for some operations.

Instances of this class are thread safe.

## Connection String

The easiest way to connect to MongoDB is to use a connection string. The standard MongoDB connection string format is a URL in the following format:

mongodb://[username:password@]hostname[:port][/database]

Note that the database name is optional.

## Create factory method

The best way to get an instance of MongoServer is to use the Create factory method. This method will return the same instance of MongoServer whenever the same connection string is used, so you don’t have to worry about creating a whole bunch of instances if you call Create more than once. Furthermore, if you are only working with one database you may find it easier to skip calling this method altogether and call the Create factory method of MongoDatabase instead.

To connect to MongoDB on localhost you would write code like:

string connectionString = “mongodb://localhost”;

MongoServer server = MongoServer.Create(connectionString);

The Create method in MongoServer ignores the database name if present. However, if the database name is omitted but credentials are present, then MongoServer will assume that these credentials are to be used with all databases and will store them in MongoServer’s Credentials property and use them as the default credentials whenever GetDatabase is called without credentials. That makes it real easy to use the same credentials with all databases.

## SafeMode property

This property represents the default SafeMode for this server. It will be inherited by any instances of MongoDatabase that are returned by this server instance, but a database’s default SafeMode can be changed indepently of the server’s after that. The SafeMode class is described further on.

## GetDatabase method

From an instance of MongoServer you can get instances of objects representing the databases on that server using the GetDatabase method. The C# driver also provides an indexer you can use if you prefer that notation. To get a database object you would write:

MongoDatabase test = server.GetDatabase(“test”);

or

MongoDatabase test = server[“test”];

The two forms are completely equivalent. If you are using authentication you have to write slightly different code, as in:

MongoCredentials credentials =

new MongoCredentials(“username”, “password”);

MongoDatabase test = server.GetDatabase(“test”, credentials);

MongoServer maintains a table of MongoDatabase instances by database/credential combinations, and each time you ask for the same database/credentials combination you get the same instance of MongoDatabase back, so you don’t need to worry about unneeded duplicate instances coming into existence.

# MongoDatabase class

This class represents a database on a MongoDB server. Unless you are using authentication, you would normally have one instance of this class for each database you want to work with. If you are using authentication then you need an instance of this class for every database/credentials combination you want to work with.

Instance of this class are thread safe.

However, if you are writing a multi-threaded program and doing a sequence of related operations (which is common!), you probably want to be using RequestStart/RequestDone to ensure that related operations on a thread all occur on the same connection to the server.

## Create factory method

Normally you get instances of MongoDatabase by calling GetDatabase on the server instance, but MongoDatabase also has a Create factory method which takes a URL as a parameter. In this case though, the database name is no longer optional, and an exception will be thrown if it is missing from the URL. If you choose to use the Create factory method in MongoDatabase you do not have to call the Create factory method in MongoServer (although it will be called for you and the resulting server object is available to you if needed in the new database instance’s Server property).

To create an instance of MongoDatabase using the Create factory method you would write:

string connectionString = “mongodb://localhost/test”;

MongoDatabase test = MongoDatabase.Create(connectionString);

MongoServer server = test.Server; // if needed

Note that this is completely equivalent to the following code:

string connectionString = “mongodb://localhost/test”;

MongoServer server = MongoServer.Create(connectionString);

MongoDatabase test = server.GetDatabase(“test”);

The first form can be advantageous if you don’t need immediate access to the server object and also because the database name (and credentials if present) come from the URL and are not hardcoded.

## Server property

This property allows you to navigate back from a MongoDatabase object to the MongoServer object that it belongs to.

## Credentials property

This property allows you to examine the credentials associated with this instance of a MongoDatabase. You cannot change the credentials. If you want to work with different credentials you have to go back to the server object and get a new database object with the new credentials. This is because an instance of MongoDatabase is normally shared by many parts of the code, and changing the credentials could have unintended side effects.

## SafeMode property

This property represents the default SafeMode for this database. It will be inherited by any instances of MongoCollection that are returned by this database instance, but a collection’s default SafeMode can be changed indepently of the database’s after that. The SafeMode class is described further on.

## GridFS property

This property gives you access to the GridFS object associated with this database. See the description of the MongoGridFS class below. The MongoGridFS class has methods like Upload, Download and Find which allow you to interact with the GridFS file system.

## GetCollection method

From an instance of MongoDatabase you can get instances of objects representing the collections on that database using the GetCollection method. The C# driver also provides an indexer you can use if you prefer that notation. To get a collection object you would write:

MongoCollection books = database.GetCollection(“books”);

or

MongoCollection books = database[“books”];

The two forms are completely equivalent.

MongoDatabase maintains a table of collections that have been used so far, and each time you ask for the same collection you get the same instance back, so you don’t need to worry about unneeded duplicate instances coming into existence.

## GetCollection<T> method

One of the features of collections in MongoDB is that they are schema-free. However, it is also not unusual for a collection to hold only one kind of document. Normally when you query a collection you have to specify the C# class that the returned documents will be deserialized into (see the Find<T> method and variants below). When you use this method the collection object returned is an instance of MongoCollection<T>, which is a subclass of MongoCollection that adds Find methods with the document type defaulting to T. This makes writing calls to Find more compact.

The initial version of the C# driver only supports returning BsonDocuments.

To use this method you would write:

MongoCollection<BsonDocument> books =

database.GetCollection<BsonDocument>(“books”);

## DropCollection method

This method can be used to drop a collection from a database. MongoCollection also has a method called RemoveAll that drops the data but leaves the collection (and any indexes) in place. To use this method write:

BsonDocument result = database.DropCollection(“books”);

DropCollection is actually a wrapper for a database command. All methods that wrap a command return the command result, which in some cases may have information of interest to you.

## RequestStart/RequestDone methods

It often happens that a thread invokes a series of database operations that are related. More importantly, it sometimes does a read that is dependent on earlier writes. If the read happens to be done on a different connection than the write sometimes the results will not yet be available. A thread can indicate that it is doing a series of related operations by using RequestStart and that it is done with the series of related operations by calling RequestDone. For example:

database.RequestStart();

// invoke a series of operations on database

database.RequestDone();

There is actually a slight problem with this example: if an exception is thrown while invoking the operations on the database RequestDone will never be called. To prevent that you could put the call to RequestDone in a finally block, or even easier, take advantage of the fact that RequestStart returns an instance of a helper object that implements IDisposable, and use the C# using statement to guarantee that RequestDone is called automatically. In this case, you write:

using (database.RequestStart()) {

// invoke a series of operations on database

}

Note that in this case you do NOT call RequestDone yourself, that happens automatically when you leave the scope of the using statement.

RequestStart increments a counter and RequestDone decrements it, so you can nest calls to RequestStart and RequestDone and the request isn’t actually over until the counter reaches zero again. This helps when you are implementing methods that need to call RequestStart/RequestDone but are in turn called by code that has itself called RequestStart. An example of this is the Upload method in MongoGridFS, which calls RequestStart/RequestDone. Because of the nesting behavior of RequestStart it is perfectly fine to call Upload from code that has itself called RequestStart.

# MongoCollection class

Instances of this class repesent a collection in a MongoDB database. You get an instance of a MongoCollection by calling the GetCollection or GetCollection<T> methods of MongoDatabase.

Instances of this class are thread safe.

## Database property

This property allows you to navigate back from a MongoCollection object to the MongoDatabase object that it belongs to.

## SafeMode property

This property represents the default SafeMode for this collection. It will be the default SafeMode for operations performed on this collection, but many methods provide a way to override the SafeMode for just that one operation. The SafeMode class is described further on.

## Insert<T> method

Before we can retrieve any information from a collection we need to put it there. The Insert method is one way to insert documents into a collection (but see also the Save method and the upsert versions of the Update method). For example:

BsonDocument book = new BsonDocument {

{ “author”, “Ernest Hemingway” },

{ “title”, “For Whom the Bell Tolls” }

};

books.Insert(book);

The insert statement could also have been written as:

books.Insert<BsonDocument>(book);

but this is not necessary as the compiler can infer the type T from the argument. Note that T must be a type that can be serialized to a BSON document. The initial version of the C# driver only knows how to serialize instances of the BsonDocument class. There is more information later in this tutorial about the various ways you can create instances of BsonDocument. The syntax used in this (and most other) examples is based on C#’s collection initializer syntax.

The Insert method can insert more than one document at once. For example:

BsonDocument book1 = new BsonDocument {

{ “author”, “Kurt Vonnegut” },

{ “title”, “Cat’s Cradle” }

};

BsonDocument book2 = new BsonDocument {

{ “author”, “Kurt Vonnegut” },

{ “title”, “Slaughterhouse-Five” }

};

books.Insert(book1, book2);

The advantage of inserting multiple documents at once is that it minimizes the number of network transmissions. All the documents being inserted are transmitted to the server in one message (unless the total size of the serialized documents exceeds 16MB, in which case they will be transmitted in multiple messages each holding as many documents as possible).

## Find<T> method

The most common operation you will perform against a collection is to query it. This is done with the several variations of the Find method. In this tutorial we only show the simplest forms, but there are other forms that let you use a JavaScript where clause instead of a query object and/or specify the fields you want returned.

Here’s some sample code to read back some of the documents we just inserted into the books collection:

BsonDocument query = new BsonDocument {

{ “author”, “Kurt Vonnegut” }

};

MongoCursor<BsonDocument> cursor =

books.Find<BsonDocument>(query);

foreach (BsonDocument book in cursor) {

Console.WriteLine(

“{0} by {1}”,

book.GetString(“title”),

book.GetString(“author”)

);

}

Unlike for the Insert method, the Find method requires us to specify the type of document being queried (BsonDocument in the above example, which is the only type supported in the initial version of the C# driver). However, if you used GetCollection<BsonDocument> to get the collection object, then it would have an overload of the Find method defaulting to BsonDocument.

There are many forms of query objects that are supported, but they are not specific to the C# driver, and we don’t have space to describe them here. An empty query object retrieves all documents, and the C# driver allows null as equivalent to an empty query object.

There is much more to say about MongoCursor, but there is a whole section below about using cursors so look there for more details.

If your collection object is of type MongoCollection<T>, then it will have a non-generic Find method which will default the return document type to T. In that case you could write:

MongoCursor<BsonDocument> cursor = books.Find(query);

And the Find method would default to the type T of the collection.

## FindAll<T> method

This method is just a shortcut for Find<T>(null) and retrieves all documents from a collection. For example:

MongoCursor<BsonDocument> cursor =

books.FindAll<BsonDocument>();

foreach (BsonDocument book in cursor) {

Console.WriteLine(

“{0} by {1}”,

book.GetString(“title”),

book.GetString(“author”)

);

}

If your collection object is of type MongoCollection<T>, then it will have a non-generic FindAll method which will default the return document type to T. In that case you could write:

MongoCursor<BsonDocument> cursor = books.FindAll();

And the FindAll method would default to the type T of the collection.

## FindOne<T> method

If you know there is only one matching document or if you only want the first matching document you can use the FindOne method. For example:

BsonDocument query = new BsonDocument {

{ “author”, “Kurt Vonnegut” }

};

BsonDocument book = books.FindOne<BsonDocument>(query);

FindOne returns either the first matching document it finds or C# null if there are none. If ther e is more than one matching document it is not specified which one will be returned.

FindOne<T> is actually a shortcut for:

Find<T>(query).Limit(1).FirstOrDefault();

Which is useful to know if you think there may be multiple matching documents and want to control which one is returned. See the example for the FirstOrDefault method of MongoCursor below.

If your collection object is of type MongoCollection<T>, then it will have a non-generic FindOne method which will default the return document type to T. In that case you could write:

BsonDocument book = books.FindOne(query);

And the FindOne method would default to the type T of the collection.

## Save<T> method

The Save method is a combination of Insert and Update. It examines the document provided to it and if the document is lacking an “\_id” element it assumes it is a new document and calls Insert on it. If the document has an “\_id” element it assumes it is probably an existing document and calls Update on it but sets the Upsert flag just in case so the document will be inserted if it doesn’t already exist. For example, you could correct an error in the title of a book using:

BsonDocument query = new BsonDocument {

{ “author”, “Kurt Vonnegut” }

{ “title”, “Cats Craddle” }

};

BsonDocument book = books.FindOne<BsonDocument>(query);

if (book != null) {

book[“title”] = “Cat’s Cradle”;

books.Save(book);

}

Note that we check the return value of FindOne to verify that it actually found a matching document.

## Update<T> method

The Update method is used to update existing documents (but see the Upsert flag below). The code sample shown in the Save method could also have been written as:

BsonDocument query = new BsonDocument {

{ “author”, “Kurt Vonnegut” }

{ “title”, “Cats Craddle” }

};

BsonDocument update = new BsonDocument {

{ “$set”, new BsonDocument { “title”, “Cat’s Cradle” } }

};

BsonDocument updatedBoook = books.Update(query, update);

The Update method supports many different kinds of update documents, but again, we don’t have space in this tutorial to describe them. The above example used the “$set” option to update the value of one element of the matching document.

There is also an overload of Update that allows you to specify one or more UpdateFlags. The values are Upsert and Multi. Normally an Update only affects one document, but using UpdateFlags.Multi you can request that all matching documents be updated. Also, by specifying UpdateFlags.Upsert you can request that the update document be inserted if a matching document can’t be found (note that the Upsert feature requires that the update document be a complete document and not just an update operation as in the sample above).

## Count method

Sometimes you just need to know how many documents a collection has. There are two versions of the count method, the first counts all documents, and the second counts matching documents. For example:

BsonDocument query = new BsonDocument {

{ “author”, “Ernest Hemingway” }

};

int total = books.Count();

int booksByHemingway = books.Count(query);

## Remove method

This method can be used to remove documents from a collection. For example, to remove all of Hemingway’s books from the collection, we write:

BsonDocument query = new BsonDocument {

{ “author”, “Ernest Hemingway” }

};

books.Remove(query);

If we only wanted to remove one book by Hemingway we would write:

BsonDocument query = new BsonDocument {

{ “author”, “Ernest Hemingway” }

};

books.Remove(query, RemoveFlags.Single);

although we wouldn’t have control over which book exactly was remove; but it would be one by Hemingway.

## RemoveAll method

If you want to remove all the documents in a collection, use:

books.RemoveAll();

Use with caution! RemoveAll is just a shortcut for Remove(null). See also the DropCollection method in MongoDatabase, although this method differs from that one in that only the data is removed, not the collection itself or any indexes.

## CreateIndex method

This method is used to create an index on a collection. In its simplest form you simply provide a list of the element names to be indexed. There are more complex forms that use a BsonDocument to specify the details of the index to be created, but they are beyond the scope of this tutorial.

To create an index for the books collection on author, we would write:

books.CreateIndex(“author”);

Note that MongoDB is forgiving if you try to create an index that already exists, but to reduce the number of times the server is asked to create the same index use the EnsureIndex method instead.

## EnsureIndex method

You will normally call this method instead of CreateIndex. The driver keeps track of which indexes have already been created and only calls CreateIndex once for each index (the driver doesn’t know whether the index actually exists on the server, so it must call CreateIndex at least once to make sure the index exists). Calling EnsureIndex is just like calling CreateIndex:

books.EnsureIndex(“author”);

## Distinct method

This method is used to find all the distinct values for a given key. For example, to find all the authors in the books collection you would write:

IEnumerable<object> authors = books.Distinct(“author”);

foreach (string author in authors) {

Console.WriteLine(author);

}

There is also an overload of the Distinct method that takes a query, so you can find the distinct values for a given key of just those documents that match the query.

## FindAndModify method

Text missing.

## Group method

Text missing.

## MapReduce method

Text missing.

# MongoCursor<T> class

Recall that the Find method doesn’t return the actual results of the find, it returns a cursor that can be enumerated to retrieve the results of the find. The query isn’t actually sent to the server until we first attempt to enumerate the cursor, which means not only that the Find method doesn’t communicate with the server (it just returns a cursor) but that we can alter the cursor in useful ways before enumerating it to control the query.

In C# the most convenient way to iterate over a cursor is to use the foreach statement. However, if necessary you can call a cursor’s GetEnumerator method and iterate over the cursor yourself manually.

Instances of this class are NOT thread safe (unlike all the classes we have described so far, which are).

## Collection property

You can use this property to navigate back from a cursor to the collection it is querying.

## Methods that modify the cursor before execution

Methods in this category are used to modify the cursor in some way before it is enumerated to control the results returned by the cursor. Note that all these methods return the cursor itself allowing them to be chained together (a “fluent interface”).

## Skip method

This method controls how many documents the server should skip over before returning results. It is often useful in pagination, although be aware that large values for Skip can become very inefficient. For example, suppose we had many books by Isaac Asimov and wanted to skip the first 5:

BsonDocument query = new BsonDocument {

{ “author”, “Isaac Asimov” }

};

MongoCursor<BsonDocument> cursor = books.Find(query);

foreach (BsonDocument book in cursor.Skip(5)) {

Console.WriteLine(book.GetString(“title”));

}

## Limit method

Similar to Skip, the Limit method lets us control how many documents are returned by the server. This example is similar to the previous one, and tells the server to return at most 10 matching documents:

BsonDocument query = new BsonDocument {

{ “author”, “Isaac Asimov” }

};

MongoCursor<BsonDocument> cursor = books.Find(query);

foreach (BsonDocument book in cursor.Limit(10)) {

Console.WriteLine(book.GetString(“title”));

}

## Sort method

Often we wish the results to be returned in a particular order. The Sort method allows us to tell the cursor what order to return the results in. In its simplest form we simply provide the names of the keys we want the results sorted on, as in:

MongoCursor<BsonDocument> cursor = books.FindAll();

foreach (BsonDocument book in cursor.Sort(“author”, “title”)) {

Console.WriteLine(

“{0}: {1}”,

book.GetString(“author”),

book.GetString(“title”)

);

}

Other overloads of the sort method let you control the sort order of each key individually.

## Other cursor modification methods

There are more cursor modification methods available that won’t be described here, but their names alone suggest their use: AddOption, Batch, Fields, Flags, Hint, and Snapshot.

## Methods that trigger enumeration

The following methods all trigger enumeration of the cursor. This means that all modifications to the cursor using the methods above must be done first, before the trigger is enumerated.

## GetEnumerator method

This method returns an IEnumerator<T> that can be used to iterate over the results. Usually you won’t call this method yourself, but will instead use the C# foreach statement to process the results, and the compiler will generate the code necessary to get and iterate over the enumerator.

## Count method

If you don’t need the individual results but just want to know the count of matching documents you can use this method. This method is equivalent to calling Count on the collection. Note that this method ignores any Skip or Limit options that have been set on the cursor (but see the Size method).

## Size method

This method is similar to the Count method, but honors any Skip or Limit options that have been set on the cursor.

## FirstOrDefault method

Sometimes you only want or expect one result, in which case you can use this method, which is more convenient than enumerating the cursor. In simple cases you might prefer to use the FindOne method in MongoCollection, but the FirstOrDefault method on the cursor lets you provide interesting options on the cursor before retrieving the one document. For example, suppose you wanted to find the first book Hemingway published:

BsonDocument query = new BsonDocument {

{ “author”, “Ernest Hemingway” }

};

MongoCursor<BsonDocument> cursor = books.Find(query);

cursor.Sort(“publicationDate”);

BsonDocument firstBook = cursor.FirstOrDefault();

Because cursor modification methods support a fluent interface, you could also chain the calls together as in:

BsonDocument query = new BsonDocument {

{ “author”, “Ernest Hemingway” }

};

BsonDocument firstBook = books.Find(query)

.Sort(“publicationDate”)

.FirstOrDefault();

## ToArray method

This method enumerates the cursor for you and returns an array of T. For example:

BsonDocument query = new BsonDocument {

{ “author”, “Ernest Hemingway” }

};

BsonDocument[] books = books.Find(query).ToArray();

If there are no matching document the return value will be an empty array, not C# null.

## ToList method

This method enumerates the cursor for you and returns List<T>. For example:

BsonDocument query = new BsonDocument {

{ “author”, “Ernest Hemingway” }

};

List<BsonDocument> books = books.Find(query).ToList();

If there are no matching document the return value will be an empty List, not C# null.

## Consuming a cursor safely

When enumeration is triggered on a cursor the cursor gets a connection from the connection pool and hangs on to it until it is no longer needed. It is important to make sure that the connection gets returned to the connection pool by ensuring that the cursor gets a chance to know that its work is done and that it can release the cursor. The implementation of MongoCursor always releases the connection as soon as it can (possibly even before enumeration of the cursor is complete) to minimize the chances of the connection being abandoned. If the cursor is not properly terminated your program will not crash, but since the connection is not returned to the connection pool it will be lost (and eventually closed when it is garbage collected) and your program will open more connections than necessary.

If a cursor is enumerated until the end you can be sure the connection was released back to the connection pool (in fact, it will have been returned as soon as a reply to the Query or GetMore message indicated there were no more results by returning a cursorId of 0). In addition, the following MongoCursor methods all guarantee that the cursor is disposed properly: Count, FirstOrDefault, ToArray and ToList.

So really the only situation you need to worry about is when you abandon a cursor that still has an open cursor at the server without cleaning it up. To help prevent this situation MongoCursor implements the IDisposable interface, and you can call Dispose to terminate the cursor early. A common way to do this is to use the cursor in a C# using statement:

using (MongoCursor<BsonDocument> cursor = books.FindAll()) {

foreach (BsonDocument book in cursor) {

// process the book

}

}

It is harmless to call Dispose when it is not needed, so the above example works fine whether the using statement exits normally after the cursor is consumed all the way to the end (and has already closed down cleanly) or whether an exception is thrown.

## Explain method

This method is different from any of the other cursor methods because it doesn’t actually return the results of the query. Instead, the server returns information about how the query would have been executed. This information can be useful when trying to figure out why a query is not performing well. To use this method you would write:

BsonDocument query = new BsonDocument {

{ “author”, “Ernest Hemingway” }

};

MongoCursor<BsonDocument> cursor = books.Find(query);

BsonDocument explanation = cursor.Explain();

# MongoGridFS class

This class is the C# driver implementation of the GridFS specification for storing files in a MongoDB database. You normally get an instance of this class by using the GridFS property of MongoDatabase.

Instances of this class are thread safe. However, if you plan to modify the Settings for this instance, you should do so before the multiple threads start accessing this instance.

## Database property

This property allows you to navigate back from the instance of MongoGridFS to the database it is associated with.

## Settings property

This property gives you access to the Settings for this instance. You can change the Root name of the GridFS collection (default “fs”) and the DefaultChunkSize for new files (default 256KB). Note that the default chunk size only applies to new files. Any files previously created with a different chunk size will continue to be processed using the existing chunk size. It is not possible to change the chunk size of a file once it has been created.

## Upload method

This method uploads a file from the client computer to the database. It always creates a new file in the GridFS file system, so if you upload the same file more than once there will be multiple versions of the file in the database. To use this method write:

MongoGridFS gridFS = database.GridFS;

MongoGridFSFileInfo fileInfo = gridFS.Upload(“volcano.jpg”);

The return value of Upload is information about the file that was just uploaded. You can ignore the return value if you wish. Upload will throw an exception if it fails.

There are other overloads of Upload that let you specify different local and remote filenames or to upload directly from a Stream instead of from a file.

## Download method

This method downloads a file from the database to the client computer. If the file already exists on the client computer it will be overwritten. To use this method write:

MongoGridFS gridFS = database.GridFS;

gridFS.Download(“volcano.jpg”);

As mentioned before, there may be multiple versions of the same filename in the database. Download by default downloads the most recent version (as defined by the “uploadDate” element of the file metadata). You can download a different version by providing an integer version parameter. The values for a version number can be:

1 The first version

2 The second version

n The nth version

-1 The newest version

-2 The second newest version

-n The nth newest version

0 Any version (not very useful unless there’s only one)

So for example, to retrieve the second newest version of our file you would write:

MongoGridFS gridFS = database.GridFS;

gridFS.Download(“volcano.jpg”, -2); // second newest version

Other overloads of the Download method let you specify different local and remote filenames or to download directly to a Stream instead of a file.

## Find method

The Find method queries the GridFS file system for information about a file. If more than one file matches, then information for all of them is returned (even when providing a specific filename there may be multiple matches if the same filename has been uploaded more than once). For example:

MongoGridFS gridFS = database.GridFS;

List<MongoGridFSFileInfo> files = gridFS.Find(“volcano.jpg”);

If there are no matches the list will be empty (not C# null).

## FindOne method

If you expect there to only be one matching filename or are willing to specify a version number then it is more convenient to use the FindOne method. For example:

MongoGridFS gridFS = database.GridFS;

MongoGridFSFileInfo file = gridFS.FindOne(“volcano.jpg”);

Or to get information about the second version of the file:

MongoGridFS gridFS = database.GridFS;

MongoGridFSFileInfo file = gridFS.FindOne(“volcano.jpg”, 2);

In both cases, if there is no matching file C# null is returned.

## Exists method

If you just want to know if the file exists but don’t need any of the metadata about the file you can use this method. For example:

MongoGridFS gridFS = database.GridFS;

if (gridFS.Exists(“volcano.jpg”)) {

// now you know the file exists

}

## Delete method

This method deletes matching files from the GridFS file system. If more than one file matches, they will all be deleted (see the sample below for how to delete just one version of a file). For example:

MongoGridFS gridFS = database.GridFS;

gridFS.Delete(“volcano.jpg”);

If you only want to delete the second newest version write something like:

MongoGridFS gridFS = database.GridFS;

MongoGridFSFileInfo fileInfo = gridFS.FindOne(“volcano.jpg”, -2);

if (fileInfo != null) {

gridFS.Delete(fileInfo.Id); // delete by \_id

}

# MongoGridFSFileInfo class

This class represents information about a file stored in the GridFS file system. This class is designed to be as similar as possible to .NET’s FileInfo class to minimize the learning curve.

## ChunkSize property

The chunk size used when this file was created.

## GridFS property

You can use this property to navigate from a MongoGridFSFileInro instance back to the MongoGridFS object it belongs to.

## Id property

This property has the value of the “\_id” element for this file. This value is assigned when the file is first uploaded and is guaranteed to be unique.

## Length property

The total length of the file in bytes.

## MD5 property

The MD5 hash of the file as computed at the server when it was uploaded.

## Name property

The remote filename of the file in the GridFS file system.

## UploadDate property

The DateTime when this file was uploaded. This DateTime is always in UTC.

## Methods to be implemented soon

This class also will implement methods that return a Stream-like object (a subclass of Stream) that allow you to read and write to GridFS files the same way you do local files (this is an alternative to using the Upload/Download methods). These methods all return an instance of MongoGridFSStream, which is a read-write stream that can also seek. The methods that return a MongoGridFSStream (or a StreamWriter wrapped around it) are: AppendText, Create, CreateText, Open, OpenRead, and OpenWrite.

# SafeMode class

Several times earlier in this tutorial we have alluded to SafeMode. There are various levels of SafeMode, and this class is used to specify those levels. SafeMode applies only to operations that don’t already return a value (so it doesn’t apply to queries or commands). It applies to the following methods of MongoCollection: Insert, Remove, Save and Update. It also applies to a MongoGridFS object as a whole, so all GridFS operations are performed at the same SafeMode level.

The gist of SafeMode is that after an Insert, Remove, Save or Update message is sent to the server it is followed by a GetLastError command so that the driver can verify that the operation succeeded. In addition, when using replica sets there are SafeModes that let us specify how many replications we want to wait to have completed before GetLastError returns. The standard way to access SafeModes is to use static properties and methods of the class, as in:

SafeMode.False

SafeMode.True

SafeMode.WaitForReplications(n);

The value of “n” includes the primary, so to wait for at least one slave to have completed the replication we would use a value of “2”.

We also mentioned that SafeMode is inherited from server to database to collection and finally to operation. This inheritance happens at the moment the corresponding object is created but can be modified indepently thereafter. So for example, if I want SafeMode to be on in general (the default is off for performance reasons) but I have one collection that I want it to be turned off for I can write:

MongoServer server = MongoServer.Create(connectionString);

server.SafeMode = SafeMode.True; // default to SafeMode.True

MongoDatabase test = server[“test”]; // inherits SafeMode.True

MongoCollection log = test[“log”]; // inherits SafeMode.True

log.SafeMode = SafeMode.False; // turn off SafeMode for log

BsonDocument importantMessage = new BsonDocument {

// contents of important message

};

log.Insert(importantMessage, SafeMode.True); // override SafeMode

The last line of code illustrates that SafeMode can be overridden at the individual operation level.

# BsonElement class

A BsonElement consists of a named and typed value. A BsonDocument (and a BsonArray) consists of a collection of BsonElements. Often the BsonElements are created automatically so you don’t have to call the BsonElement constructor yourself.

Note that BsonElement implements Equals and GetHashCode, so BsonElements can be used as keys in Dictionaries.

## BsonElement constructor

There are two BsonElement constructors. In the first version the BsonType of the new element is implied by the type of the value provided (although not all types are valid, and if the value provided can’t be mapped to a BsonType an exception will be thrown). In the second version you specify the BsonType you want and an attempt will be made to convert the value provided to that BsonType (again, an exception will be thrown if the conversion is not possible). Here are some examples:

BsonElement e1 = new BsonElement(“e1”, 1);

BsonElement e2 = new BsonElement(“e2”, 2.0);

BsonElement e3 = new BsonElement(“e4”, 2, BsonType.Double);

BsonElement e4 = new BsonElement(“e5”, Guid.NewGuid());

The first example implies a BsonType of Int32 and the second of Double. The third example explicitly specifies a BsonType of Double, and the integer value 2 will be converted to double. The third example will create a value that is an instance of BsonBinaryData with a sub type of Uuid.

Note that the value of BsonElement can never be C# null, not even when you want a BSON null in the document. The BSON Null, MaxKey and MinKey types each have one possible value represented by a singleton of the BsonNull, BsonMaxKey and BsonMinKey classes respectively. To create a BSON element with one of these values you write:

BsonElement e1 = new BsonElement(“name”, Bson.Null);

BsonElement e2 = new BsonElement(“name”, Bson.MaxKey);

BsonElement e3 = new BsonElement(“name”, Bson.MinKey);

Another BSON type that requires special handling is the Symbol type. Since the C# language (or the .NET Framework) do not provide native support for symbols, BSON symbols are represented using the BsonSymbol class and are managed by the BsonSymbolTable class. To create a BSON element with a symbol as a value use:

BsonSymbol symbol = BsonSymbolTable.Lookup(“value”);

BsonElement e = new BsonElement(“name”, symbol);

or:

BsonElement e = new BsonElement(

“name”,

BsonSymbolTable.Lookup(“value”)

);

## Name property

The name of the element.

## Value property

The value of the property returned as type object. You can downcast this to the actual type if you know what it is. Note that the Value property will never be C# null (a value of BsonType.Null will be represented as the singleton instance of the BsonNull class).

## BsonType property

The BSON type of this element as a value of the BsonType enumeration. The value of this property determines what the class of the Value is.

## BsonType to .NET Type mapping

The following table defines the .NET type of the Value property for each value of the BsonType enumeration:

|  |  |
| --- | --- |
| **BsonType** | **.NET Type** |
| Double | double |
| String | string |
| Document | BsonDocument |
| Array | BsonArray |
| Binary | BsonBinaryData |
| ObjectId | BsonObjectId |
| Boolean | bool |
| DateTime | DateTime (converted to UTC) |
| Null | BsonNull (a singleton) |
| RegularExpression | BsonRegularExpression |
| JavaScript | BsonJavaScript |
| Symbol | BsonSymbol |
| JavaScriptWithScope | BsonJavaScriptWithScope (a subclass of BsonJavaScript) |
| Int32 | int |
| TimeStamp | BsonTimestamp |
| Int64 | long |
| MinKey | BsonMinKey (a singleton) |
| MaxKey | BsonMaxKey (a singleton) |

## .NET Type mapping to BsonType

Any of the .NET Types in the right hand column of the previous table are trivially mapped to their corresponding BsonType. In addition, the BsonElement constructor provides for mapping several common .NET Types to the closest matching BsonType (with conversions as needed):

|  |  |
| --- | --- |
| **.NET Type** | **BsonType** |
| Guid | BsonBinaryData (subtype Uuid) |
| byte | Int32 |
| byte[] | BsonBinaryData (subtype General) |
| DateTimeOffset | DateTime (converted to UTC) |
| float | Double |
| Regex | BsonRegularExpression |
| sbyte | Int32 |
| short | Int32 |
| uint | Int32 |
| ushort | Int32 |
| ulong | Int64 |
| IDictionary<string, object> | BsonDocument |
| IEnumerable<object> | BsonArray |

Many of the above .NET Types can also be coerced to other BsonTypes by using the three argument version of the BsonElement constructor but we will not list all the possible combinations here (basically if you think the conversion is possible it probably is).

# BsonDocument class

This class is the workhorse of the C# driver. You will use it a lot! Because it is used so much the interface is a little bit complicated, in order to provide for the many common use cases. There are basically four ways to create and populate a BsonDocument:

1. Create a new document and call Add and Set methods
2. Create a new document and use C# collection initializer syntax
3. Create a new document and use the fluent interface style supported by Add
4. Create a new document and use XElement style functional construction

The first is straightforward and easy to use, but the second, while requiring you to be familiar with collection initializer syntax is much more readable and is the recommended way to create BsonDocuments.

## BsonDocument constructor

You can use the BsonDocument constructor to create an empty BsonDocument or as a quick way to create a BsonDocument with a single element. All overloads of BsonDocument except two simply call the matching Add method with the arguments provided.

To create an empty BsonDocument write:

BsonDocument document = new BsonDocument();

To create a document and populate it with one element write:

BsonDocument query = new BsonDocument(“author”, “Hemmingway”);

By default BsonDocument does not allow duplicate element names, and if you attempt to Add an element with an existing name an exception is thrown. If you want to allow duplicate element names you have to use the following constructor:

BsonDocument document = new BsonDocument(true); // allow dups

document.Add(“hobby”, “hiking”);

document.Add(“hobby”, “cycling”); // only acessible via index

string h1 = document.GetString(“hobby”); // returns “hiking”

string h2 = document.GetString(0); // also returns “hiking”

string h3 = document.GetString(1); // returns “cycling”

There are also overloads of the BsonDocument constructor that take a:

* BsonDocument
* BsonElement
* IDictionary<string, object>
* IEnumerable<BsonElement>
* IEnumerable<object>
* params BsonElement[]
* params object[]

See the corresponding Add method for a description of each one.

## BsonDocument constructor with collection initializer syntax

This is the preferred way to create and populate BsonDocuments (with the exception of single element documents which are more easily created by passing name and value to the constructor). The way C# collection initializer syntax works is that it lets you provide a list of values (some of which can be tuples) that are matched to corresponding Add methods. The compiler simply inserts calls to the corresponding add method. Many of the BsonDocuments created in sample code you’ve seen so far were created using this syntax. An example should illustrate:

BsonDocument book = new BsonDocument {

{ “author”, “Ernest Hemingway” },

{ “title”, “For Whom the Bell Tolls” }

};

This is converted by the compiler to the following:

BsonDocument book = new BsonDocument();

book.Add(“author”, “Ernest Hemingway”);

book.Add(“title”, “For Whom the Bell Tolls”);

The two are completely equivalent, but the first is more elegant and readable because it mirrors the structure of the BsonDocument being created and is less verbose.

A common mistake when using collection initializer syntax is to forget one of the sets of braces. The reason this doesn’t work is apparent when we see that the compiler will convert:

BsonDocument wrong = new BsonDocument { “name”, “value” };

to:

BsonDocument wrong = new BsonDocument();

wrong.Add(“name”);

wrong.Add(“value”);

which somewhat unfortunately compiles without error because it matches one of the overloads for the Add method that exists to support functional construction style. Even though this error does not result in a compile time error it will result in a runtime exception being thrown that should alert you to what has gone wrong.

## BsonDocument constructor with fluent interface Add methods

Another way to construct a BsonDocument is to use the fluent interface style Add methods to populate the document. For example:

BsonDocument book = new BsonDocument()

.Add(“author”, “Ernest Hemingway”)

.Add(“title”, “For Whom the Bell Tolls”);

We don’t recomment this style because we feel C# collection initializer syntax is superior, but if you are porting code from another driver you may find this style useful.

## BsonDocument constructor with functional construction

This way of constructing BsonDocuments is particulary suited to upcoming LINQ support, and is modeled after XElement's functional construction. For example:

BsonDocument book = new BsonDocument(

new BsonElement(“author”, “Ernest Hemingway”),

new BsonElement(“title”, “For Whom the Bell Tolls”)

);

In this case the BsonDocument will be calling the Add(IEnumerable<BsonElement>) overload of the add method. One of the useful aspects of functional construction that is patterned after XElement is that any C# null values passed in are ignored. This makes it possible to use functional construction even when some of the elements are optional; just make sure the value at that position is null and it will be ignored.

## Count property

This property returns the number of elements that this document contains.

## Elements property

This property returns an IEnumerable<BsonElement> value that can be used to enumerate over the elements of the document.

## Names property

This property returns an IEnumerable<string> value that can be used to enumerate over the names of the elements of the document.

## Values property

This property returns an IEnumerable<object> value that can be used to enumerate over the values of the elements of the document.

## Add methods

There are many overloads of the Add method. Note that in all cases C# null values are ignored, so you don’t have to put if statements around your calls to Add to check if your value is null (assuming you want it to be ignored). When Add is passed a C# null value it simply does nothing.

## Add(BsonElement) method

This is the base Add method. All other Add methods end up calling this one. This method behaves slightly differently depending on whether duplicate element names are allowed or not. If duplicate names are not allowed and an element with the same name exists then an exception is thrown, otherwise it is added to the end of the collection and can only be accessed by index (accessing by name will return the already existing element, not the new one).

## Add(IEnumerable<BsonElement>) method

This method simply calls Add(BsonElement) for each element passed in.

## Add(name, value) method

This method creates and adds one BsonElement to the document. For example:

BsonDocument book = new BsonDocument();

book.Add(“author”, “Ernest Hemingway”);

Since this Add method passes its arguments straight through to the BsonElement constructor you can refer to the discussion above about creating BsonElements and how the value provided is mapped to a BsonType.

## Add(name, value, bsonType) method

This method is similar to the previous Add method but you control the resulting BsonType. If the value provided cannot be converted to that BsonType an exception will be thrown.

BsonDocument book = new BsonDocument();

book.Add(“price”, 20, BsonType.Double);

## Add(condition, name, value) method

This method will add a new BsonElement with name and value if the condition is true. This method exists to support conditionally adding elements to a document when using C# collection initializer syntax. For example:

BsonDocument book = new BsonDocument {

{ “author”, “Ernest Hemingway” },

{ “title”, “For Whom the Bell Tolls” },

{ havePublicationDate, “publicationDate”, publicationDate }

};

This is only really necessary for struct values (because they can’t be C# null). For example, suppose you only wanted to add the “subtitle” element if it was not C# null. We could simply write:

BsonDocument book = new BsonDocument {

{ “author”, “Ernest Hemingway” },

{ “title”, “For Whom the Bell Tolls” },

{ “subtitle”, subTitle }

};

If the value of the subTitle is C# null the “subtitle” element will not be added to the document.

## Add(condition, name, value, bsonType) method

This method is similar to the previous method but lets you control the BsonType of the element. If the value cannot be converted to that BsonType an exception is thrown.

## Add(BsonDocument) method

Each element of the BsonDocument argument will be added to this BsonDocument. Note that the elements are not cloned, so now the same BsonElement is in both BsonDocuments. This means that if you change the value of the BsonElement in one docucment it will change in the other as well. If you don’t want this behavior then you must clone the elements before adding them to your document. For example:

BsonDocument order = new BsonDocument();

BsonDocument items; // passed in from somewhere

order.Add(items);

or if you don’t want to share the elements between the two documents use one of:

order.Add(items.Clone());

order.Add(items.DeepClone());

depending on whether you need shallow or deep cloning.

The default behavior is not too clone because cloning is expensive and this way we default to the most efficient behavior and you decide when the cost of cloning is necessary.

## Add(IDictionary<string, object>) method

This method adds new elements to a BsonDocument based on the contents of a dictionary. Each key in the dictionary becomes the name of the new element, and each corresponding value become the value of the new element. As always, this method calls Add(BsonElement) with each new element, so duplicate names are handled as described in Add(BsonElement).

## Add(params BsonElement[]) method

This overload is provided as a convenience to allow providing a variable number of BsonElement arguments in the calling code. It simply delegates to Add(IEnumerable<BsonElement>).

## Get methods

There are a number of get methods that return the value of an element. The name of the method indicates what .NET Type the value will be returned as (or in the case of the GetAs methods converted to). These methods can throw exceptions if the value is not of the correct type. Each one of these methods is provided in two forms: one takes a numeric index and the other the element name. Here is a list of the get methods provided:

* GetArray
* GetAsBoolean (using JavaScript’s definition of truthiness)
* GetAsDouble
* GetAsInt32
* GetAsInt64
* GetBinaryData
* GetBoolean
* GetDateTime
* GetDouble
* GetElement
* GetGuid
* GetInt32
* GetInt64
* GetJavaScript
* GetJavaScriptWithScope
* GetMaxKey
* GetMinKey
* GetNull
* GetObjectId
* GetRegularExpression
* GetString
* GetSymbol
* GetTimestamp

Here are just a few examples:

BsonDocument book = books.FindOne();

string author = book.GetString(“author”);

DateTime publicationDate = book.GetDateTime(“publicationDate”);

int pages = book.GetInt32(“pages”);

BsonDocument also provides an indexer to get element values, but in this case the return type is always object and you will have to cast it yourself. The previous example could be written using indexers as:

BsonDocument book = books.FindOne();

string author = (string) book[“author”];

DateTime publicationDate = (DateTime) book[“publicationDate”];

int pages = (int) book[“pages”];

## Merge method

This method allows you to merge elements from one BsonDocument into another. Each element from the source document is tested to see if the target document already has an element with that name. If the name already exists, the element is skipped, if not it is added to the target document.

## RemoveElement method

This method is used to remove an element from a BsonDocument. If the document allows duplicate names then a call to RemoveElement will remove all elements with that name.

## RemoveElementAt method

This method is used to remove an element from a BsonDocument using its index position.

## Set(index, value) method

While most of the time you will access elements by name, occassionally you will access them by index. When using this method the document must already have an element at that index or an exception will be thrown.

## Set(index, value, bsonType) method

This method is like the previous one but you control the resulting BsonType. If value cannot be converted to bsonType an exception will be thrown.

## Set(name, value) method

If an element of this name exists, its value is replaced with the new value, otherwise a new element with this name and value are added. For example:

BsonDocument book = new BsonDocument();

book.Set(“author”, “Ernest Hemway”); // adds element

// some other work

// notice that author name is spelled wrong

book.Set(“author”, “Ernest Hemingway”); // replaces value

BsonDocument also provides an indexer to get and set element values. The previous example could be written using indexers as:

BsonDocument book = new BsonDocument();

book[“author”] = “Ernest Hemway”; // adds element

// some other work

// notice that author name is spelled wrong

book[“author”] = “Ernest Hemingway”; // replaces value

## Set(name, value, bsonType) method

This method is like the previous one but you control the resulting BsonType. If value cannot be converted to bsonType an exception will be thrown.

## ToBson method

This method will serialize the BsonDocument to the binary BSON format and return the serialized document as a byte array.

## ToJson method

This method will serialize the BsonDocument as a JSON formatted string. It is possible to have some control over the resulting JSON representation by providing some BsonJsonWriterSettings.

## ToString method

This method is overridden to call ToJson using default settings. It is primarily useful while debugging. If you are intentionally creating JSON call ToJson instead.

# BsonArray class

This class is used to represent BSON arrays. Note that while a BSON array is represented externally as a BSON document (with a special naming convention for the elements), in the C# driver the BsonArray class is NOT related to the BsonDocument class. That is because the fact that a BsonArray is represented externally as a BsonDocument is accidental, and the actual behavior of a BsonArray is very different from that of a BsonDocument.

## BsonArray constructors

There are several constructors that vary only in whether and how they add elements to the array at the time it is constructed. Some samples are probably self explanatory:

BsonArray a1 = new BsonArray(); // empty

BsonArray a2 = new BsonArray(1); // 1 element

BsonArray a3 = new BsonArray(1, BsonType.Double); // 1 element

BsonArray a4 = new BsonArray(1, 2, 3) // 3 elements

The values added to a BsonArray are just like the values added to a BsonElement. The same rules apply for converting .Net Types to BsonTypes.

## Indexer property

This class provides an indexer property to get or set elements in the array. The indexer return type is object, so you have to do any casting yourself. For example:

array[0] = “Tom”;

array[1] = 39;

string name = (string) array[0];

int age = (int) array[1];

## Count property

This property returns the number of values stored in the BsonArray

## Add methods

There are several Add methods, differing in how they add elements to the array. Most of them are self explanatory:

BsonArray array = new BsonArray();

array.Add(1); // 1 element

array.add(1, BsonType.Double); // 1 element

array.Add(1, 2, 3); // 3 elements

In addition there is an overload that takes an IEnumerable<object> value and simply adds all the objects to the array:

IEnumerable<object> values = …; // from somewhere

array.Add(values);

## Clear method

This method clears all values from the array.

## Insert method

This method is used to add a value at a particular location in the array. For example:

array.Insert(2, value);

This example assumes that the array contains at least 2 elements. If it contains more than two elements, the elements starting at index 2 will be moved over one position to make room for the new element.

## Remove method

The Remove method removes a particular value from the array. If the value occurs more than once, all instances are removed. All remaining elements are shifted to the right so as to leave no gaps in the array and the resulting array will be shorter (unless no matching elements were found in which case this method has no effect).

## RemoveAt method

This method removes a value from a particular location in the array. Any elements following it are moved one position to the left and the array is now one element shorter.

# BSON value classes

There are several classes whose purpose is to hold values of BSON types that can’t be represented using standard .NET Types. They were alluded to earlier when we discussed type mapping in the description of BsonElement.

These classes are rather self explanatory, so we will simply provide a number of examples here:

byte[] bytes = …;

BsonBinaryData binary = new BsonBinaryData(bytes);

BsonBinaryData guid = new BsonBinaryData(Guid.NewGuid());

string code = “function …”;

BsonJavaScript js = new BsonJavaScript(code);

string code = “function …”;

BsonDocument scope = new BsonDocument {

{ “x”, 1 },

{ “y”, 2 }

};

BsonJavaScriptWithScope jsws = new BsonJavaScriptWithCode(

code,

scope

);

byte[] bytes = …; // some 12 byte value

BsonObjectId oid1 = new BsonObjectId(bytes);

BsonObjectId oid2 = BsonObjectId.GenerateNewId();

DateTime creationTime = oid2.CreationTime;

int machine = oid2.Machine;

int pid = oid2.Pid;

int increment = oid2.Increment;