

WebData - Week 2

▼ Mongo DB Indexing & Data Calculations

Indexes Theory

- Refer To: ***Database Indexes Simplified*** handout.

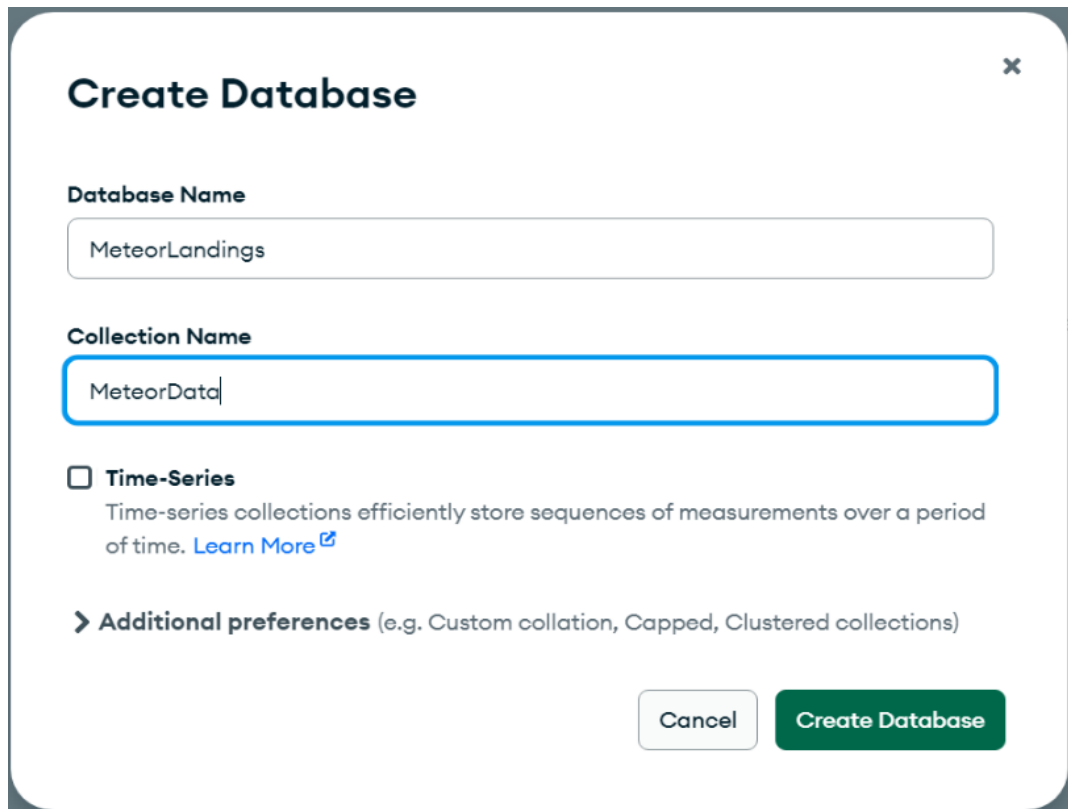
▼ Setting Indexes in MongoDB

Setup

- Download ***Meteor_Landings_Data*** File From Connect
 - A CSV file tracking know meteorite landings on earth

Open and Start a New Database in Mongo DB

- Open Mongo
- Create New Database
 - DB Name - MeteorLandings
 - Collection - Meteor Data

A modal dialog box titled "Create Database" with a close button (X) in the top right corner. It contains two input fields: "Database Name" with the text "MeteorLandings" and "Collection Name" with the text "MeteorData". Below these is a checkbox labeled "Time-Series" which is unchecked. A descriptive text below the checkbox says "Time-series collections efficiently store sequences of measurements over a period of time. [Learn More](#)". Below that is a link "Additional preferences" followed by "(e.g. Custom collation, Capped, Clustered collections)". At the bottom right are two buttons: "Cancel" and "Create Database".

Create Database

Database Name
MeteorLandings

Collection Name
MeteorData

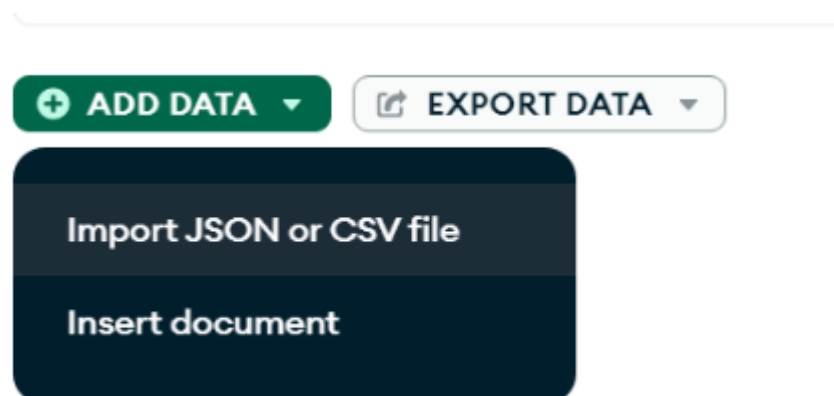
☐ **Time-Series**
Time-series collections efficiently store sequences of measurements over a period of time. [Learn More](#)

> **Additional preferences** (e.g. Custom collation, Capped, Clustered collections)

Cancel Create Database

Use CSV Upload tool to upload file data

- Click the Add Data Button,
- Select the **Import JSON or CSV file** option.



- Select the **Meteor_Landings_Data** File from it's file location.
- The next screen will show the first 10 or so entries and make data type suggestions based upon it's analysis of all the provided data.
 - Check the suggested data types are suitable and press **Import** to add the file's data.

Import

To collection MeteorLandings.MeteorData

Select delimiter Comma

☒ Ignore empty strings

☐ Stop on errors

Specify Fields and Types [Learn more about data types](#)

	<input checked="" type="checkbox"/> name String	<input checked="" type="checkbox"/> id Int32	<input checked="" type="checkbox"/> nametype String	<input checked="" type="checkbox"/> recclass String	<input checked="" type="checkbox"/> mass (g) Number
1	Aachen	1	Valid	L5	21
2	Aarhus	2	Valid	H6	720
3	Abee	6	Valid	EH4	107000
4	Acapulco	10	Valid	Acapulcoite	1914

Cancel Import

- Your data should now load into your Database Collection.
- Repeat the process 2-3 times with the same file so we have more data for testing the next steps.

MeteorLandings.MeteorData137.1k 3
DOCUMENTS INDEXES

Documents Aggregations Schema Indexes Validation

Filter ⓘ ⓘ Type a query: { field: 'value' } or [Generate query](#) ⚡

Explain Reset Find ⌕ Options ▶

➕ ADD DATA 📄 EXPORT DATA ✎ UPDATE 🗑 DELETE

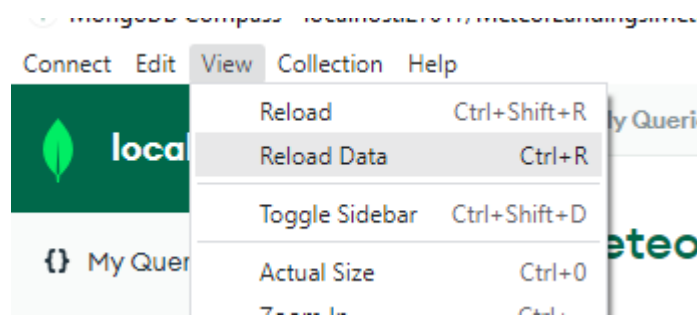
1 - 20 of 9645 ⌂ ⏪ ⏩ ⌵ ⌶ ⌷

```
_id: ObjectId('65ceb0d6d3fd5f9312e79be6')
name: "Bates Nunataks 00306"
id: 4964
nametype: "Valid"
recclass: "L5"
mass (g): 66.95
fall: "Found"
year: 2000
reclat: -80.25
reclong: 153.5
GeoLocation: "(-80.25, 153.5)"
```

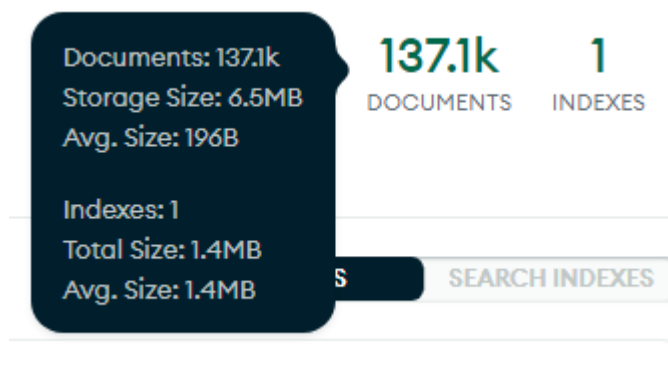
```
_id: ObjectId('65ceb0d6d3fd5f9312e79be8')
name: "Bates Nunataks 00308"
id: 4966
nametype: "Valid"
recclass: "L5"
mass (g): 10.7
fall: "Found"
year: 2000
reclat: -80.25
reclong: 153.5
GeoLocation: "(-80.25, 153.5)"
```

Testing the Data in Mongo DB Without Indexes

- Once data is uploaded review collection data.
- Go to the View > Reload Data option in the main menu to refresh the display.



- Once refreshed hover your mouse over the document count at the top right of the main tab.
- This will show the details of how much data is being used to store your collection.



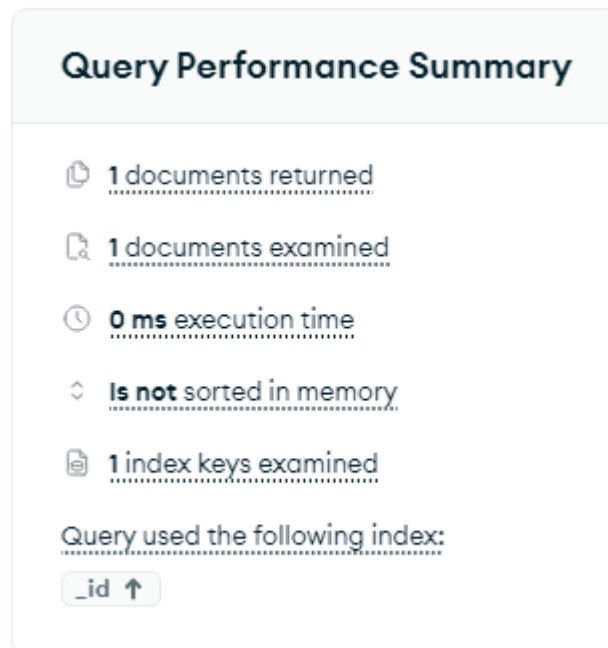
- Look at size of data used by documents compared to indexing for just the `_id` field
 - Index is equivalent to 25-30% of the data size
- Show search using `_id` field in query - Pick entry from later in collection
 - For example:

```
{_id:ObjectId('64d06543c4ed19a4e67a0ad4')}
```

- Note the speed of retrieval - almost instant
- Click Explain Button to the right of the filter text box



- This will open a window to let you analyze the efficiency of the filter when it was executed.



- Because the `_id` field is indexed we can note the following:
 - Note only 1 entry was examined or retrieved
 - This is because the location was found in the index and only matched one entry
 - Almost 0ms used to execute query
- Now Search On A different Field using the Filter Field
 - Query Name field.



- Click The Explain Button once the search finishes
 - The number of documents examined will be the total document count
 - Check execution time on right hand side will be much larger

- Compare the differences

Query Performance Summary






- 1 documents returned
- 45716 documents examined
- 21 ms execution time
- Is not sorted in memory
- 0 index keys examined
- No index available for this query.



- Note time to retrieve - Write down
 - May look quick but watch refresh icon on entry count under find buttons
- Now perform a filter/search using both the recclass and year fields at the same time
-

```
{ $and: [{ recclass: "L5" }, { year: { $gte: 2000 } } ] }
```

- Go to Explain button Note time to execute and write it down
 - This will return more documents in total but still needs to evaluate every one in the collection.

Query Performance Summary

-  **9645** documents returned
-  **137148** documents examined
-  **69 ms** execution time
-  **Is not** sorted in memory
-  **0** index keys examined

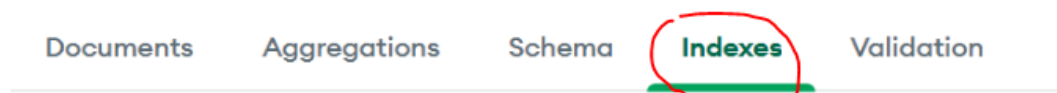
 **No index available for this query.** 

▼ Add Indexes to Check Improvement

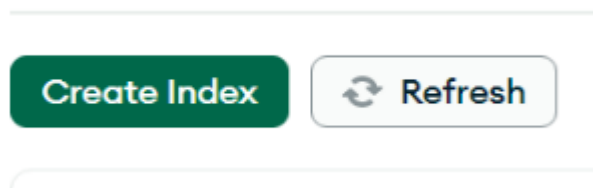
Create Single Field Index

- Go to the Indexes Tab on the Collection screen.

MeteorLandings.MeteorData



- Click the Create Index Button



- Fill out the index form to select the name field and set an **1asc**(ascending) index type.

Create Index ✕

MeteorLandings.MeteorData

Index fields

name ▼

1 (asc) ▼

+

> Options

Cancel

Create Index

- - Run the name check again and compare results in the explain tab

Query Performance Summary

- 📄 1 documents returned
- 🔍 1 documents examined
- 🕒 0 ms execution time
- ⚡ Is not sorted in memory
- 📄 1 index keys examined

Query used the following index:

name ↑

- Note Number of documents examined now the same as the retrieved value
- The time is much faster

Create Compound index for recclass and year fields

- Open Index Tab Again and open new index tab
- Select both year and recclass fields and assign ascending index for both.

- Use + sign on first field to add 2nd field.

×

Create Index

MeteorLandings.MeteorData

Index fields

recclass	1 (asc)	+	–
year	1 (asc)	+	–

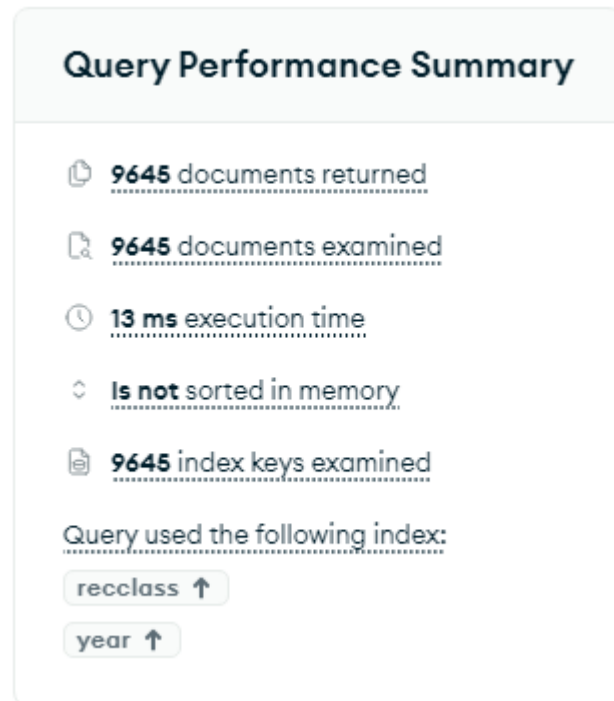
> Options

Cancel
Create Index

- Run year and recclass search again

```
{ $and: [{ recclass: "L5" }, { year: { $gte: 2000 } } ] }
```

- Now check the results with the Explain button.
 - Should be faster
 - Won't quite be as fast as a single value search.



Adding Add A Text Index

An alternative to regular indexes for text fields in Mongo DB is what is known as a text index. A text index allows you to specify several text fields like a compound index but instead of an ascending or descending type you select the text option.

This option is best suited for documents where there are several text fields and you want to be able to search all the fields at the same time for a certain keyword. This is great for situations where there are several strings in the entry that are somewhat related and when you need to search you are more likely to check all the fields instead of an individual one.

Setting a Text Index

- Open the index page and choose the **name** and **recclass** fields and assign them a text index type.

Create Index



MeteorLandings.MeteorData

Index fields

name	▼	text	▼	+	-
recclass	▼	text	▼	+	-

> Options

Cancel

Create Index

NOTE: You can only have 1 text index per collection.

Searching On A Text Index

When you search against a text index you don't need to specify each field being searched. You just use the special **\$text** operator in your filter then add the required parameters.

- You format the filter using the parameters shown below
 - You can leave some sections off if you want
 - The only mandatory field is the **\$search** parameter.

```
{
  $text:
  {
    $search: <string>,
    $language: <string>,
    $caseSensitive: <boolean>,
    $diacriticSensitive: <boolean>
  }
}
```

Where would you use a text filter?

Imagine if you were to have a series of documents in your database about products, and each one had a title, a brief description and then a

detailed description all stored as strings in the entry.

If you want to search for a keyword or term related to the product, there is a chance it may show up in any of these fields. A text index would allow you to run a specialized search using this index which would search all the fields at once instead of you having to enter an individual filter for each field.

The Benefits/Trade-offs for Indexes

- **Benefits**

- Increased query speed & retrievals
 - Finds required documents faster
 - Doesn't need to check entire collection
- More operations per minute on common searches
 - Faster retrieval means more operations in same timeframe
- Less request backlog
 - More operations per minute means less waiting and queueing.
- Querying multiple fields using text indexes
 - Can search multiple fields at once without writing separate filters.

- **Trade-offs**

- Size of indexes in memory
 - Indexing can take more memory than the initial data storage
 - Increases overall storage requirements
- Costs for insertion/updates - loss of insertion speed
 - Every insert now needs to be indexed and index tables need to be re-sorted
 - More operations per insert or update

▼ Time To Live (TTL)

When indexing on datetime fields you also have the option to apply a TTL or Time To Live parameter.

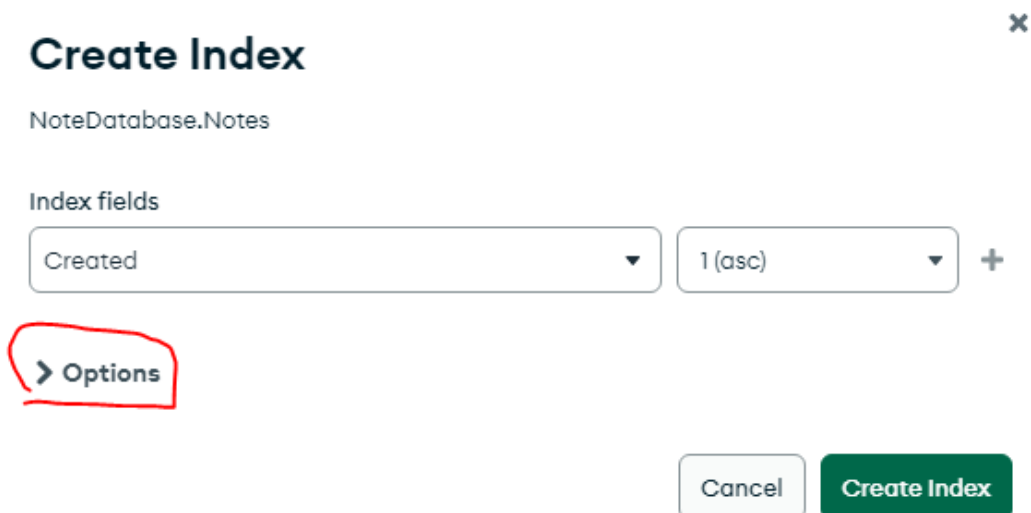
A TTL sets the database to mark the entry for deletion once the data in the specified field has passed a certain point in real time.

For example if a date in the document is dated 12:00pm 12/02/2030 and we set a TTL on this field for 6 months, the record will be deleted once our real time is 6 months past the date stored in the data field. In this example that would mean that the record would be flagged for deletion at 12:00pm 12/09/25, which is 6 months past the initial date.

The deletion would then occur within the next 60-90 seconds alongside any other records that have been flagged for deletion at that time.

Setting A TTL

- Start setting your index like normal on any field that is a date time data type.
- Then press the options toggle to open the advanced menu options.



Create Index ×

NoteDatabase.Notes

Index fields

Created ▼ 1 (asc) ▼ +

> Options

Cancel Create Index

- In this menu, tick the checkbox next to the **Create TTL** option and it will open a text field.

Create Index



NoteDatabase.Notes

Options

☐ **Create unique index**

A unique index ensures that the indexed fields do not store duplicate values; i.e. enforces uniqueness for the indexed fields.

☐ **Index name**

Enter the name of the index to create, or leave blank to have MongoDB create a default name for the index.

☒ **Create TTL**

TTL indexes are special single-field indexes that MongoDB can use to automatically remove documents from a collection after a certain amount of time or at a specific clock time.

seconds

☐ **Partial Filter Expression**

Partial indexes only index the documents in a collection that meet a specified filter expression.

Cancel

Create Index

- In this field specify the amount of time (in seconds) you want to set the TTL for.
- Below are some examples of values you might use:
 - 1 day = 86400 sec
 - 1 week = 604800 sec
 - 30 days = 2592000 sec
 - 60 days = 5184000 sec
 - 365 days = 31536000 sec
- Once finished you can press **Create Index** to complete the Index and TTL

As you can see, calculating months or weeks to seconds can be a bit annoying for some time-frames. An alternative to this, you can have a field which specifies the date when you want the entry to be deleted and then set the TTL to trigger after about 10 seconds.

This can often be easier as your programming language might provide better features/methods to set these dates initially in your code before sending the data to MongoDB

▼ Calculating Storage & Throughputs

Sometimes as part of managing and maintaining your Mongo DB data store you might want to calculate the average storage increase over time to help plan for the potential need for upsizing/scaling up the system.

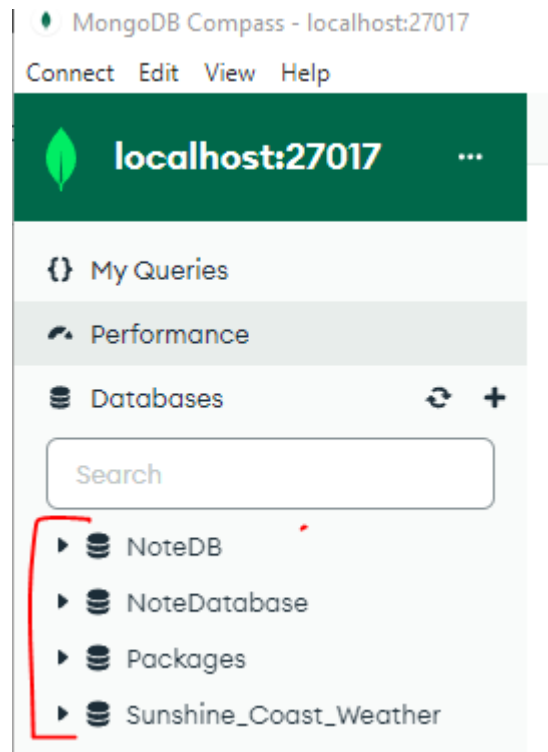
You may also need to determine the average volume of data being inputted and outputted to and from the system.

We are now going to look at some simple ways of determining this using the values available in the Mongo Compass interface which will need to be used alongside our knowledge for the system's average usage.

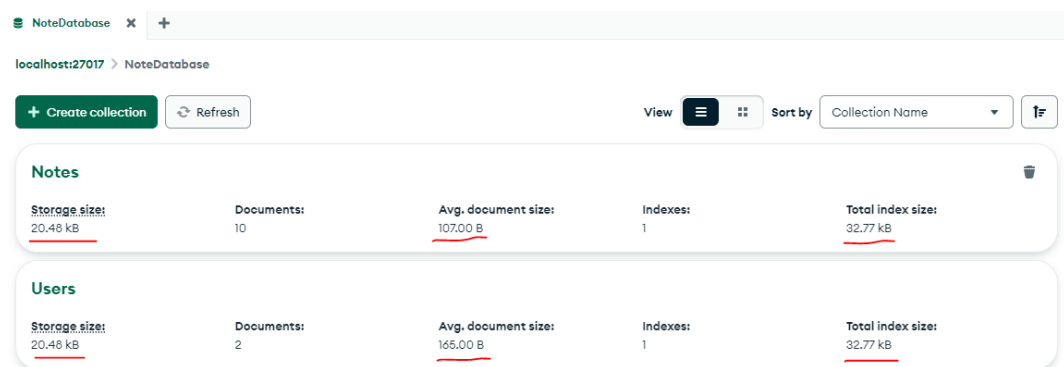
▼ Calculating Storage

For most calculations the first thing you will need is to know the current storage and the average document size in your collection/s.

The best way is to start by clicking on the database name in your list of databases in the left hand section of Mongo DB compass.

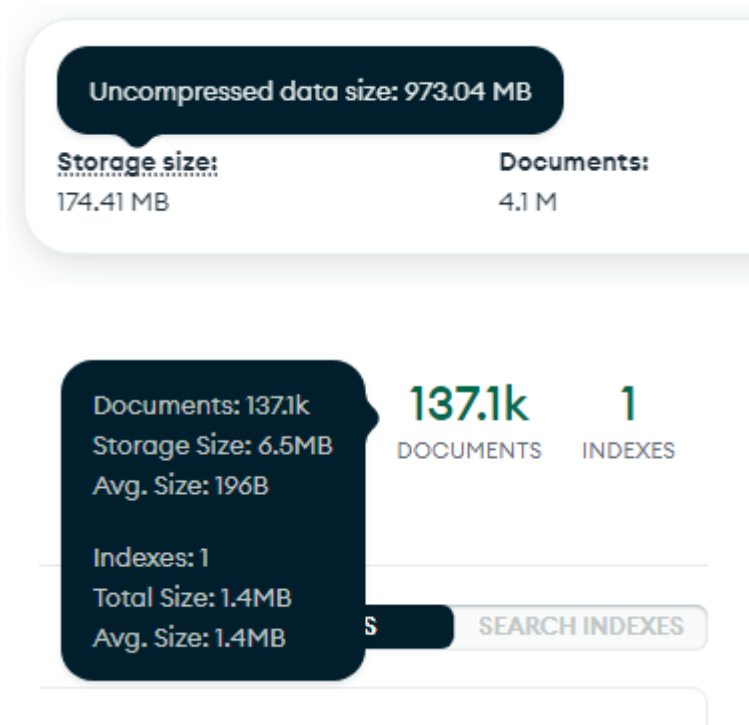


Once this is done it will open the database details in the main window section and it will display each collection in your database as well as the stats about each collection.



This will show the current storage values for your data and indexes as well as the average document size of all your documents in the collection.

Be aware that the storage sizes shown here for the total storage will be the compressed data size, not the original size. To get the initial size prior to compression you need to hover your mouse over the Storage Size field.



You can also sometimes get the indexes and average documents size when you have the collection open and looking in the top right of the compass UI in some versions. This section will not allow you to view the uncompressed document storage value though.

NOTE: Be aware that if you have very little documents in the database the uncompressed data size will seem smaller than the final compressed value, this is because Mongo DB appears to have a minimum storage size which is probably used for setting up the collection management and storage system, even when it is basically empty. As you add enough data to meet the minimum, this inconsistency will fix itself.

Calculating Initial Data Size (Before compression)

Once you have these values you can easily calculate the uncompressed storage size growth of your document store by multiplying the average document size by the expected number of insertions over time.

For example, if our average document size is 107 bytes and we are expecting a fairly consistent input rate of 20 documents per day, we can use these to calculate the overall growth of the collection.

By multiplying 20 per day by 365 days in a year we would get an expected input of 7300 documents per year.

By multiplying this by the average document size of 107 bytes we would get a total of 781,100 bytes of data expected over the year.

To convert this to kilobytes we simply divide it by 1000 (1024 if you want to be more accurate) which would be 781.1 kb.

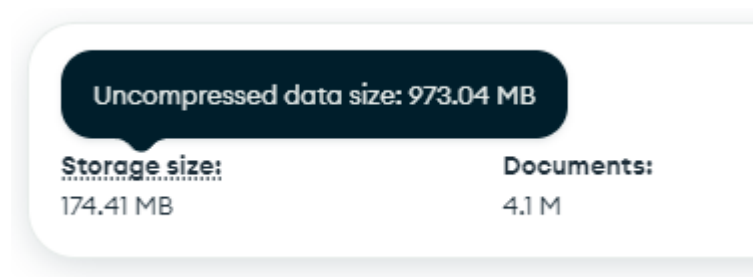
If we want to convert it to Mb we simply repeat the same calculation again which would get us to 0.7811 Mb in a year.

Once we have this value we can add it to our current storage value to get our total size after 1 year. Let's assume our already existing data was 0.55 Mb, then by adding our increase value we would end up with a total collection size of 1.3311 Mb after a year.

Calculating After Compression Size

If you want to calculate what this growth size will be after compression you need to work out the rate of compression being achieved in your documents. This will vary slightly between collections depending on the data types used.

To get this rate, start by looking at your existing storage values for both the storage size and the uncompressed storage size.



To work out the average rate of compression your documents are getting, simply divide the uncompressed value by the compressed result, this will give you a guide on how much it is being compressed when being stored.

For example, if I was to use the values in the image above, I would divide my starting document storage size (973.04 Mb) by the final compressed size (174.41 Mb).

$$973.04 / 174.41 = 5.5790378$$

If I was to round this to 3 decimal places I would get a rough compression rate of 5.579 to 1.

To apply this to my growth calculations I would then just divide my expected final collection size after adding one year of growth by my value of 5.579 to get my approximate size after being compressed. This would give me my expected storage size on the hard disk to store my new entries.

▼ Calculating Throughputs

To calculate your expected data throughputs you need to start by retrieving your average document size like we did in the last section.

After you have this you need to know what your average number of input and output requests are going to be to your system. In an already running system, this might be able to be done by using a tool to log or monitor the number of requests being made to your database.

In many cases though, you will likely have to discuss with your client and analyse the intended usage of the system to work out the expected average usage of the system.

This can be easy if the data is coming from a consistent source and is updating/adding values at a fixed rate.

Or if the client's business already has a rough idea of how many interactions they do with the system on average based upon their previous business statistics.

For example, if the system is recording chemical readings in a warehouse and has 5 sensors logging fresh reading every 2 minutes, you can calculate the average number of readings per hour or per day.

*60 mins / 2 minutes * 5 sensors = 300 entries / hour*

If the client has an store where they serve an average of 5 customers per hour and each customer requires the staff to query 2 collections to check user details and write to a 3rd collection for each customer per sale then you can use these to get your approximate values.

*5 * read to collection 1 / hour*

*5 * read to collection 2 / hour*

*5 * write to collection 3 / hour*

It may take some analysis or require you and your client to make some assumptions to determine these values but once you do you can use these values to calculate your final averages.

Once you have worked out an average figure for your expected inputs and outputs, you simply need to multiply these values by the average document size for the collection that is expected to be written to or read from.

For example, if we were to continue our sensor reading scenario from above and after checking our average document size we find out that each reading is an average of 126 bytes, we can then multiply this by the number of documents we calculated would be input in an hour (300) to get our final hourly rate.

300 records per hour * 126 bytes per document = 37,800 bytes / hour

In our second example above, because we are interacting with 3 different collections you would just do this calculation separately for each collection, using the different average sizes. Once this is done you would add the inputs together and the outputs together to get your separate input and output values.

For example if our 3 collections had the average document sizes:

Collection 1 - 56 bytes / document

Collection 2 - 132 bytes / document

Collection 3 - 95 bytes / document

Then our averages per collection would be:

Collection 1 - 5 reads per hour * 56 bytes = 280 bytes / hour

Collection 2 - 5 reads per hour * 132 bytes = 660 bytes / hour

Collection 3 - 5 writes per hour * 95 bytes = 475 bytes / hour

Once we add any reads and writes together

Writes = 475 bytes / hour

Reads = 940 bytes / hour

Once these values are calculated in either example you just need to divide the size by 1000 to get from bytes to kilobytes and then by 60 to get from hours to minutes then by 60 again to get from minutes to seconds.

For example if we were to convert the sensor example to kb/sec we would start with our original rate of:

37,800 bytes / hour

Then to get to kb we would divide by 100:

37,800 / 1000 = 37.8 kb / hour

The to convert to seconds:

37.8 / 60 min / 60 seconds = 0.0105 kb / sec

This would mean our sensor system is inputting an average of 0.0105 kb / sec of data being input into the system.

This same calculation can be applied to the per hour values of any of our example calculations.

As long as our hardware and database server is capable of maintaining this input rate, we should be able to run it consistently without needing to upgrade our system. If we start to increase the size of our system and are expecting your input and output rates to increase, then we will need to recalculate the updates inputs and decide if we need to upgrade our system.

▼ Continuing On our MongoDB API

▼ Add Our Note Data Model

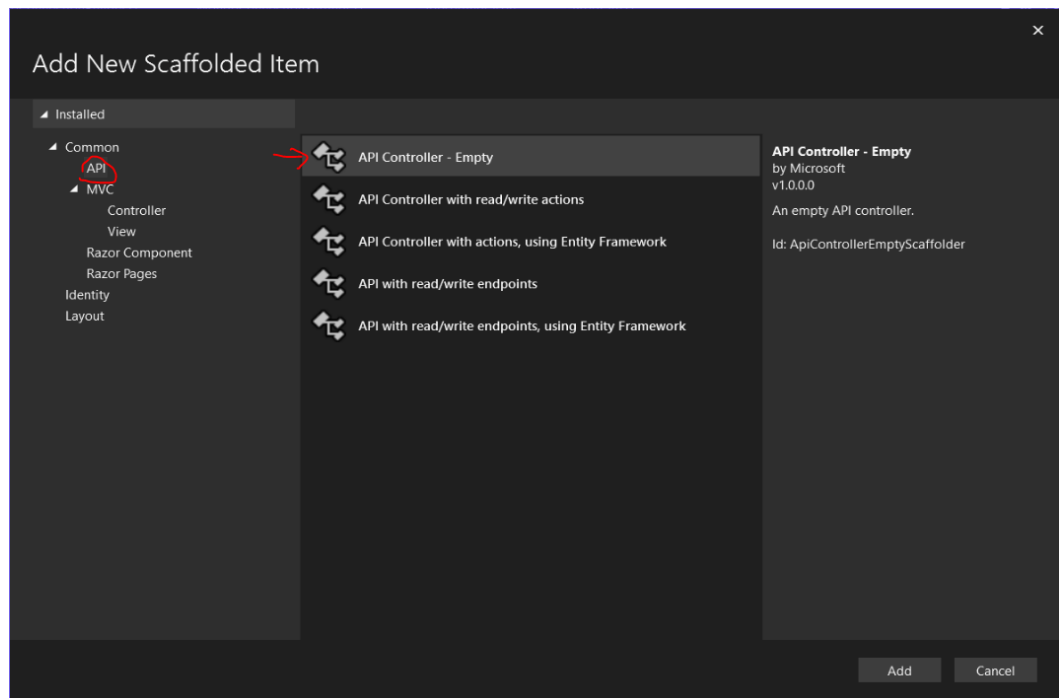
- Create Models folder and build model for note
 - The id is set as an ObjectId instead of a string due to the way MongoDB sets ids.
 - The id field is names as '_id' as this is the MongoDB naming convention.

```
0 references
public class Note
{
    [BsonId]
    0 references
    public ObjectId _id { get; set; }
    0 references
    public string Title { get; set; }
    0 references
    public string Body { get; set; }
    0 references
    public DateTime Created { get; set; }
}
```

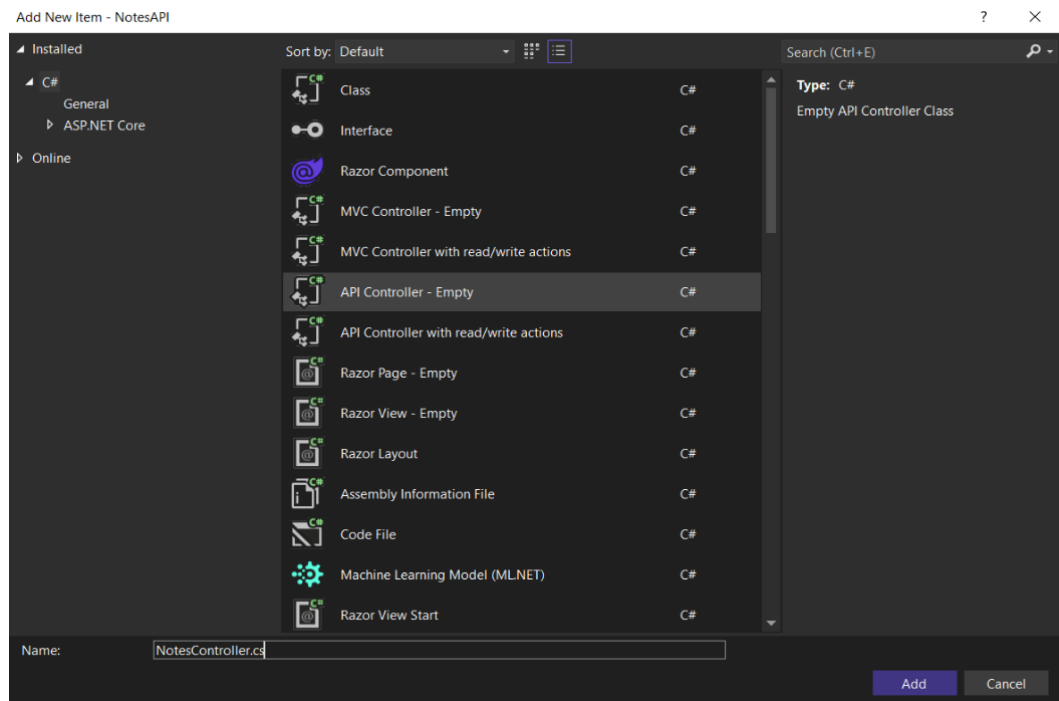
```
using MongoDB.Bson;  
using MongoDB.Bson.Serialization.Attributes;
```

▼ Create Notes Controller

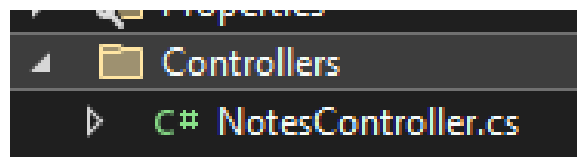
- Create NoteController to use this operation
 - Right click on Controllers folder
 - Select API controller - empty then press add.



- On the following screen name your controller **NotesController.cs** then press **ADD**



- Once finished it should appear in your Controller folder.



▼ Add Basic Note Controller Endpoints

- Start with the endpoint for the Get All Notes Functionality.
 - Name your method and tag it with the **[HttpGet]** attribute
 - Give method an ActionResult return type that holds a list of Notes.
 - Make the method return and **Ok()** response for now.

```
//GET: api/Note
[HttpGet]
0 references
public ActionResult<List<Note>> GetNotes()
{
    return Ok();
}
```

- Now let's build with the endpoint for the Get Single Note Functionality.
 - Name your method and tag it with the **[HttpGet("GetSingle")]** attribute
 - The GetSingle will modify the URL for the endpoint to require **/GetSingle** in the address.
 - Give method an ActionResult return type that holds single Notes.
 - Make the method require a string called id as an input parameter.
 - Make the method return and **Ok()** response for now.

```
//GET: api/Note/GetSingle
[HttpGet("GetSingle")]
0 references
public ActionResult<Note> GetNote(string id)
{
    return Ok();
}
```

- Now let's build with the endpoint for the Create Note Functionality.
 - Name your method and tag it with the **[HttpPost]** attribute
 - Give method a simple ActionResult return type
 - Make the method require a Note object as an input parameter.
 - Make the method return and **Ok()** response for now.

```
//POST: api/Note
[HttpPost]
0 references
public ActionResult PostNote(Note note)
{
    return Ok();
}
```


- Now let's build with the endpoint for the Update Note Functionality.
 - Name your method and tag it with the **[HttpPut]** attribute
 - Give method a simple ActionResult return type
 - Make the method require a string for an id and a Note object as input parameters.
 - Make the method return and **Ok()** response for now.

```
//PUT: api/Note
[HttpPut]
0 references
public ActionResult UpdateNote(string id, Note note)
{
    return Ok(note);
}
```

- Now let's build with the endpoint for the Delete Note Functionality.
 - Name your method and tag it with the **[HttpDelete]** attribute
 - Give method a simple ActionResult return type
 - Make the method require a string called id as an input parameter.
 - Make the method return and **Ok()** response for now.

```
//DELETE: api/Note
[HttpDelete]
0 references
public ActionResult DeleteNote(string id)
{
    return Ok();
}
```

▼ Repository Pattern Explanation

- Refer to: **Repository & Service Pattern** handout.
- Introduction to Repository Pattern:
 - Define what is a Repository Pattern

- Explain the main reason for using a Repository Pattern
- Discuss the main benefits of using a Repository Pattern

▼ Implement Repository Pattern

- Add a repository Folder to the Project
 - Add interface called INoteRepository to folder
 - This will contain all the standard CRUD operation types as part of the interface.
 - This Interface will only specify the methods that can be called upon the repository, but has no implementation.

```
public interface INoteRepository
{
    2 references
    List<Note> GetAllNotes();
    2 references
    Note GetNoteById(string id);
    2 references
    void PostNote(Note note);
    2 references
    void UpdateNote(string id, Note note);
    2 references
    void DeleteNote(string id);
}
```

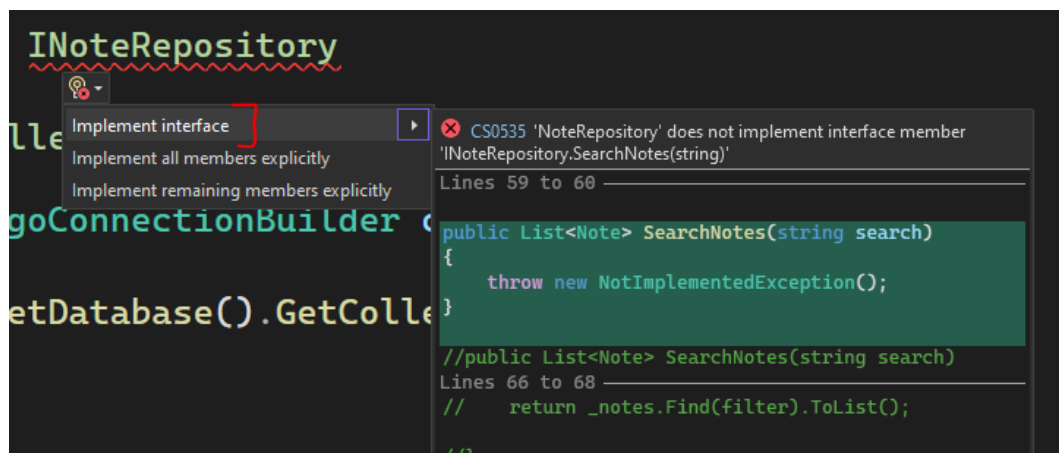
Explaining interfaces again briefly

- Outline a contractual set of methods the inheriting class must implement
- Forms a established set of agreed upon commands that can be used between components.
- Any class implementing the interface and can be replaced so long as the class replacing it used the same methods as defined by the interface.
- A class requesting something to be done via the interface it just makes the request and passes any required data. It doesn't care how it is processed.
- Create a class in the Repository folder called MongoNoteRepository

- Set the class to inherit from INoteRepository

```
0 references
public class MongoNoteRepository: INoteRepository
{
    ...
}
```

- The INoteRepository will immediately underline in red.
 - This is because the class is inheriting the interface but is not yet compliant with the required methods specified in the interface.
- Then open your **NoteRepository** class and use the lightbulb shortcut to add the new method.
 - Hover mouse over the Interface declaration
 - Click on the lightbulb icon when it appears
 - Select the **Implement Interface** option
 - The required method will build automatically.



- d
- Add constructor and readonly field to class to get Mongo Connection from dependency injection and map it to a `IMongoCollection`
- This one is a bit different
 - The readonly field is for a `IMongoCollection` which takes a type for the data to be mapped to

- The Dependency injection gets the MongoClient and passes it to the constructor.
- Inside the constructor the connection is accessed to get the desired connection and then pass it to the _notes field.

```
public class NoteRepository : INoteRepository
{
    private readonly IMongoCollection<Note> _notes;
    0 references
    public NoteRepository(MongoConnectionBuilder connection)
    {
        _notes = connection.GetDatabase().GetCollection<Note>("Notes");
    }
}
```

- Add NoteRepository class to the services in program.cs
 - This version adds the class but also defines the interface that needs to be used to request it.
 - Any controller/classes needing access to the repository now just needs to request it in their constructor.
 - NOTE: A good thing about MongoDB is that if the database does not exist, it will be built the first time you use this repository if it is setup correctly.

```
//Adds the repository class to the services by providing
//the interface so that we can request it using the interface
//type instead of the class name. This makes it easier to change
//this class for an alternative (SQL) without changing all the rest of
//our code.
builder.Services.AddScoped<INoteRepository, NoteRepository>();
```

▼ Add Repository To Controller

- Add constructor and read only field requesting access to INoteRepository

```

public class NotesController : ControllerBase
{
    private readonly INoteRepository _repository;
    0 references
    public NotesController(INoteRepository repository)
    {
        _repository = repository;
    }
}

```

▼ Add CRUD Operations

Now that our repository and controller are in place and connected to each other, we can start filling out the code in them to process each request and pass the relevant values back and forth from the database.

As we fill these out we will focus on each action one at a time and fill out the controller and endpoint for each one before moving to the next endpoint operation.

Add First CRUD operation to system - POST

We will start by completing our POST endpoint first. This will allow us to add more entries to the Database that we can use when testing the rest of the endpoints.

- Start by adding the Post method functionality to NotesController
 - This method simply passes the details on to the repository before responding to the user.
 - The Response code in the return line gives back 201 HTTP response to indicate a new resource has been created.

```

// POST api/<NotesController>
[HttpPost]
0 references
public ActionResult Post([FromBody] Note note)
{
    _repository.PostNote(note);
    return CreatedAtAction("POST", note);
}

```

- Next, we will add the required code to the Post Note method in the NoteRepository
 - Probably the easiest endpoint to do at this stage.
 - Calls the inserts method and passes it the note provided by the controller.

```
public void PostNote(Note note)
{
    //Request for the note to be saved in Mongo DB.
    _notes.InsertOneAsync(note);
}
```

Once this is done run your application and test your endpoint. The swagger UI should provide a JSON object in this section that you can fill out to add an entry. Go ahead and add a few notes so we have some data to work with for later functionality.

Add Get All functionality

Next we will complete the Get All functionality so we can retrieve and view all the records in the Notes collection..

- First, start by adding the required code to the Get method in the controller.
 - The first line forwards the request to the repository and stores the returned values in a variable.
 - The return line replies with an OK (200 response) as well as passing the results from the repository back to the user.

```
// GET: api/<NotesController>
[HttpGet]
0 references
public ActionResult<List<Note>> Get()
{
    var notes = _repository.GetAllNotes();
    return Ok(notes);
}
```

- Now, add the required code to the GetAllNotes() method in the repository.
 - The first line requests a Filter builder from the Builders class that be used to set the filter rules for the request.
 - The 2nd line uses the filter builder to create an empty filter rule.
 - All Find() requests require a filter, even if it is an empty one.
 - The final line passes the filter to the find method to process the request and then puts the result in a list before returning it to the controller.

```
2 references
public List<Note> GetAllNotes()
{
    var builder = Builders<Note>.Filter;
    var filter = builder.Empty;

    return _notes.Find(filter).ToList();
}
```

Once finished, run the API to test your endpoint.

Explain ID field as shown in API results

You may notice when you ran the Get All endpoint that the ID field actually returns a series of values instead of the unique string we see in the Mongo Db interface.

```
{
  "_id": {
    "timestamp": 1720660842,
    "machine": 214515,
    "pid": 22491,
    "increment": 14453850,
    "creationTime": "2024-07-11T01:20:42Z"
  },
  "title": "Updated Note",
  "body": "Updated using our API and swagger",
  "created": "0001-01-01T00:00:00"
},
{
```

This is because the string is actually a calculated value that is generated using these values to produce a unique value.

To make our endpoints display the Id in the string format we need to make some adjustments to the model for our note.

- Add [JsonIgnore] tag to _id field in model - this will make it ignore this field in the JSON objects in the API responses.
- This will not stop it being used for the database
- It will only be ignored in the JSON object passed to the API

```
public class Note
{
    [JsonIgnore]
    [BsonId]
    0 references
    public ObjectId _id { get; set; }
```

- Under the _id property add a new property with just a getter
 - Can use the fat arrow shorthand to do this(see below)
 - When called it will call the ToString() on the fields of the _id to get a string
 - This string output is much more readable and usable when searching the database.


```

[JsonIgnore]
[BsonId]
1 reference
public ObjectId _id { get; set; }
//Same as setting a property with just a getter
0 references
public string ObjId => _id.ToString();
2 references

```

Once this is done, run your endpoint again and see how the model is shown now. It should show the id as a string now that looks like it does in Mongo DB.

```

{
  "objId": "668f336a0345f357dbdc8c5a",
  "title": "Updated Note",
  "body": "Updated using our API and swagger",
  "created": "0001-01-01T00:00:00"
},

```

Add Get Single Functionality

Next we will setup our Get Single note endpoint.

- Start by updating the endpoint in the controller.
 - The first line forwards the request and the provided to the repository and stores the returned result in a variable.
 - The return line replies with an OK (200 response) as well as passing the result from the repository back to the user.

```

// GET api/Notes/GetSingle
[HttpGet("GetSingle")]
0 references
public ActionResult<Note> Get(string id)
{
    //Pass the request on to the repository to get all the notes
    var note = _repository.GetNoteById(id);
    //Pass the note that gets returned by the repository back to the user.
    return Ok(note);
}

```

- Next we will fill out the GetNoteById() method in the repository.
 - Even though a string for the _id is easier to use for the API, to find a match for a single entry based upon Id you need to convert the

string back to an ObjectId as per how they are stored in the database

- Once done you need to create a filter to use the ObjectId to find an exact match in the database
- Finally, pass the filter to the find method to get the record you are after.

```
public Note GetNoteById(string id)
{
    //Convert the id string back to it's proper ObjectId format.
    ObjectId objId = ObjectId.Parse(id);
    //Create a filter to compare the _id field of each record
    //against the objId variable.
    var filter = Builders<Note>.Filter.Eq(n => n._id, objId);
    //Run the request and ask for the first item that matches.
    return _notes.Find(filter).FirstOrDefault();
}
```

Add Delete functionality

- First setup the same filter as our Get single method
- Then call the delete one method

```
public void DeleteNote(string id)
{
    //Convert the id string back to it's proper ObjectId format.
    ObjectId objId = ObjectId.Parse(id);
    //Create a filter to compare the _id field of each record
    //against the objId variable.
    var filter = Builders<Note>.Filter.Eq(n => n._id, objId);
    //Run the request and ask for item to be deleted.
    _notes.DeleteOneAsync(filter);
}
```

- Update the controller

```
// DELETE api/<NotesController>/5
[HttpDelete("{id}")]
0 references
public ActionResult Delete(string id)
{
    _repository.DeleteNote(id);
    return Ok();
}
```

Add Update Functionality

- In MongoDB there are technically 2 types of update
 - Replace
 - Overwrites previous record with new record
 - Removes old content and then add new content.
 - Can lose data if new data is incomplete or has null fields
 - Update
 - Updates specified fields only
 - Requires more work to implement
 - Less risk of data loss.
- In our API we will use the Update version for better data safety.
- Setup your controller

```
// PUT api/<NotesController>/5
[HttpPut("{id}")]
0 references
public ActionResult Put(string id, [FromBody] Note note)
{
    if (String.IsNullOrEmpty(id) || note == null)
    {
        return BadRequest();
    }

    _repository.Update(id, note);
    return Ok();
}
```

- Update your Repository code
 - For Update start with the same filter as before
 - Before calling the method to do the update/replace, we need to add the provided ID to the note model.
 - Failure to do this will cause error
 - Create update rules using builder.
 - Can chain each update command into one long command.

```
public void UpdateNote(string id, Note note)
{
    //Convert the id string back to it's proper ObjectId format.
    ObjectId objId = ObjectId.Parse(id);
    //Create a filter to compare the _id field of each record
    //against the objId variable.
    var filter = Builders<Note>.Filter.Eq(n => n._id, objId);
    //Create an update rule builder for setting our details of
    //which fields we want to update.
    var builder = Builders<Note>.Update;
    //Define the changes you want made to the record. Each set
    //commands is stating which field to change and what value
    //to change it to.
    var update = builder.Set(n => n.Title, note.Title)
        .Set(n => n.Body, note.Body);
    //Send the request to MongoDB to be processed.
    _notes.UpdateOneAsync(filter, update);
}
```

-

Using the Replace Instead - NOT RECOMMENDED.

- When calling the update you can instead use the Replace command in the controller.
- This has a highwrr risk of error if you are not careful
 - Make sure all fields of the new Note are filled out.
 - Only use when you have already retrieved the full note details prior to update.

```
2 references
public void Update(string id, Note updatedNote)
{
    //Convert provided string Id back to ObjectId type
    ObjectId objId = ObjectId.Parse(id);
    //Create filter looking for exact match of ObjectId
    var filter = Builders<Note>.Filter.Eq(n => n._id, objId);
    //Pass the provided Id into the Ote Object so it matches the
    //one being replaced
    updatedNote._id = objId;
    //Pass the note and the filter to find and replace the object
    _notes.ReplaceOne(filter, updatedNote);
}
```

Recap Completed Functionality

- Once finished this should cover all the standard CRUD database operations
- For your assessment you will potentially need to add create many, update many or delete many methods
- We will also need to look at validation and error checking/messaging

Continuing Assessment Work

Keep working on your assessment with the remaining time we have in class today. You should now be able to start looking at the Indexing and Data Storage/Throughput sections of Part 1.

You could also do some more work on your API, now that we have covered the basics of creating and connecting your endpoints. You should be able to use today's example to start implementing the required endpoints in your assessment API. Be aware that not all the endpoint types we have used today will be needed in the assessment so make sure you review the assessment documents to determine which ones will and will not be needed.