- Using IAggregateFluent
- Using LINQ
- Parsing BsonDocument ( the MongoDB's shell-like syntax )

1. Download MongoDB server @ https://www.mongodb.com/try/download/community
2. Download Studio3T Client @ https://studio3t.com/download/
3. https://www.mongodb.com/products/compass

## Getting Setup

MongoDB only creates the database when you first store data in that database. This data could be a collection or even a document.

1. create a Console App (.NET Core), and nameit `MongoDBConnectionDemo`.

Next, we need to install the MongoDB Driver for C#/.NET for a Solution. We can do that quite easily with NuGet. Inside Visual Studio for Windows, by going to *Tools* -> *NuGet Package Manager* -> Manage NuGet Packages for Solution..._

We can browse for *MongoDB.Driver*. Then click on our Project and select the driver version we want. In this case, the latest stable version is 2.16.1. Then click on *Install*. Accept any license agreements that pop up and head back to Program.cs to get started.

## Putting the Driver to Work

2. To use the **MongoDB.Driver** we need to add a directive.

```
using MongoDB.Driver;
```

3. Inside the `Main()` method we'll establish a connection to Database with a connection string and to test the connection we'll print out a list of the databases on the server.

The first step is to pass in the MongoDB Atlas connection string into a MongoClient object, then we can get the list of databases and print them out.

```
using MongoDB.Driver;

MongoClient dbClient = new MongoClient("mongodb://localhost:27017");

var dbList = dbClient.ListDatabases().ToList();

Console.WriteLine("The list of databases on this server is: ");
foreach (var db in dbList)
{
    Console.WriteLine(db);
}
```

When we run the program, we get the following out showing the list of databases:

The list of databases on this server is:

```
The list of databases on this server is:
```

```
{ "name" : "admin", "sizeOnDisk" : NumberLong(40960), "empty" : false }
{ "name" : "config", "sizeOnDisk" : NumberLong(61440), "empty" : false }
{ "name" : "local", "sizeOnDisk" : NumberLong(40960), "empty" : false }
```

## Create Document

1. Add the rows to set which Database and Collection we are going to use

```
using MongoDB.Bson;
using MongoDB.Driver;

MongoClient dbClient = new MongoClient("mongodb://localhost:27017");

var database = dbClient.GetDatabase("mongo_sample");
var collection = database.GetCollection<BsonDocument>("persons");
```

## Creating a BSON Document

2. The collection variable is now our key reference point to our data.
   Since we are using a `BsonDocument` when assigning our collection variable, I've indicated that I'm not going to be using a pre-defined schema. This utilizes the power and flexibility of MongoDB's document model.
   *I could define a plain-old-C#-object (POCO) to more strictly define a schema*.
   We'll take a look at that option in a future post.
   For now, we create a new `BsonDocument` to insert into the database.

```
using MongoDB.Bson;
using MongoDB.Driver;

MongoClient dbClient = new MongoClient("mongodb://localhost:27017");

var database = dbClient.GetDatabase("mongo_sample");
var collection = database.GetCollection<BsonDocument>("persons");

var document = new BsonDocument {
    { "person_id", 10000 },
    { "name", "Lisa"},
    { "surname", "Grant"},
    { "scores",  new BsonArray {
            new BsonDocument { { "type", "exam" }, { "score", 88.1 } },
            new BsonDocument { { "type", "quiz" }, { "score", 74.9 } },
            new BsonDocument { { "type", "homework" }, { "score", 89.9 } },
            new BsonDocument { { "type", "homework" }, { "score", 82.1 } }
        }
    },
    { "class_id", 480 }
};
```

## Create Operation

3. Then to *Create* the document in the we can do an insert operation.

```
using MongoDB.Bson;
using MongoDB.Driver;
```

```
MongoClient dbClient = new MongoClient("mongodb://localhost:27017");

var database = dbClient.GetDatabase("mongo_sample");
var collection = database.GetCollection<BsonDocument>("persons");

var document = new BsonDocument {
    { "person_id", 10000 },
    { "name", "Lisa"},
    { "surname", "Grant"},
    { "scores",  new BsonArray {
            new BsonDocument { { "type", "exam" }, { "score", 88.1 } },
            new BsonDocument { { "type", "quiz" }, { "score", 74.9 } },
            new BsonDocument { { "type", "homework" }, { "score", 89.9 } },
            new BsonDocument { { "type", "homework" }, { "score", 82.1 } }
        }
    },
    { "class_id", 480 }
};

collection.InsertOne(document);
// If you need to do that insert asynchronously
// the MongoDB C# driver is fully async compatible
// The same operation could be done with:
await collection.InsertOneAsync(document);
```

- **NOTE: the first insert creates the Database and the collection | Also _id has been added to the schema**
- If you have a need to insert multiple documents at the same time, MongoDB has you covered there as well with the `InsertMany` or `InsertManyAsync` methods.

### Wrapping Up

We've seen how to structure a BSON Document in C# and then *Create* it inside a MongoDB database. The MongoDB C# Driver makes it easy to do with the `InsertOne()`, `InsertOneAsync()`, `InsertMany()`, or `InsertManyAsync()` methods

As I mentioned, using a `BSONDocument` is convenient when a schema isn't defined.

More frequently, however, the schema is defined in our code, not in the database itself.

Now that we have *Created* data, we'll want to *Read* it. I'll show that step in the CRUD process in my next post.

## Read Operations

1. To *Read* documents in MongoDB, we use the `Find()` method. This method allows us to chain a variety of methods to it, some of which I'll explore in this post. To get the first document in the collection, we can use the `FirstOrDefault` or `FirstOrDefaultAsync` method, and print the result to the console.

```
using MongoDB.Bson;
using MongoDB.Driver;

MongoClient dbClient = new MongoClient("mongodb://localhost:27017");

var database = dbClient.GetDatabase("mongo_sample");
var collection = database.GetCollection<BsonDocument>("persons");

Console.WriteLine("Here is the reading result:");
var firstDocument = collection.Find(new BsonDocument()).FirstOrDefault();
Console.WriteLine(firstDocument.ToString());
```

returns...

```
Here is the reading result:
{ "_id" : ObjectId("62bdfee7f92bc9a2cbe0ed66"), "person_id" : 10000,
"name" : "Lisa", "surname" : "Grant", "scores" : [{ "type" : "exam",
"score" : 88.1 }, { "type" : "quiz", "score" : 74.9 }, { "type" :
"homework", "score" : 89.9 }, { "type" : "homework", "score" :
82.129310305132179 }], "class_id" : 480 }
```

- You may wonder why we aren't using Single as that returns one document too. Well, that has to also ensure the returned document is the only document like that in the collection and that means scanning the whole collection.

### Reading with a Filter

Let's find the document we created and print it out to the console.

1. The first step is to create a filter to query for our specific document.

```
var filter = Builders<BsonDocument>.Filter.Eq("person_id", 10000);
```

Here we're setting a filter to look for a document where the `person_id` is equal to 10000.

2. We can pass the filter into the `Find()` method to get the first document that matches the query.

```
var studentDocument = collection.Find(filter).FirstOrDefault();
Console.WriteLine(studentDocument.ToString());
```

- If a document isn't found that matches the query, the `Find()` method returns `null`.

Finding the first document in a collection, or with a query is a frequent task. However, what about situations when all documents need to be returned, either in a collection or from a query?

### Reading All Documents

For situations in which the expected result set is small, the `ToList()` or `ToListAsync()` methods can be used to retrieve all documents from a query or in a collection.

1. Create other two items:

```
var newDocuments = new BsonDocument[]
{
    new () {
        { "person_id", 10001 },
        { "name", "Claire"},
        { "surname", "Robinson"},
        { "scores",  new BsonArray {
                new BsonDocument { { "type", "exam" }, { "score", 89.1 } },
                new BsonDocument { { "type", "quiz" }, { "score", 74.2 } },
                new BsonDocument { { "type", "homework" }, { "score", 98.9 } }
            }
        },
```

```
        { "class_id", 481 }
    },
    new()
    {
        { "person_id", 10002 },
        { "name", "Mike"},
        { "surname", "Bell"},
        { "scores",  new BsonArray {
                new BsonDocument { { "type", "exam" }, { "score", 91.1 } },
                new BsonDocument { { "type", "quiz" }, { "score", 70.2 } },
            }
        },
        { "class_id", 482 }
    }
};

await collection.InsertManyAsync(newDocuments);
```

2. Set Find to get all documents (with an empty filter).

```
var documents = collection.Find(new BsonDocument()).ToList();
```

3. We can iterate over that list and print the results like so:

```
foreach (BsonDocument doc in documents)
{
    Console.WriteLine(doc.ToString());
}
```

4. Filters can be passed in here as well, for example, to get documents with exam scores equal or above 95. The filter here looks slightly more complicated, but thanks to the MongoDB driver syntax, it is relatively easy to follow. We're filtering on documents in which inside the scores array there is an exam subdocument with a score value greater than or equal to 95.

```
var highExamScoreFilter = Builders<BsonDocument>
    .Filter
    .ElemMatch<BsonValue>(
        "scores",
        new BsonDocument {
            { "type", "exam" },
            { "score", new BsonDocument {
                    { "$gte", 90 }
                }
            }
        });
var highExamScores = collection.Find(highExamScoreFilter).ToList();

foreach (BsonDocument doc in highExamScores)
{
    Console.WriteLine(doc.ToString());
}
```

- Where there's more than a small list of results you need it retrieve, the `Find` method can be made to return a cursor with `ToCursor` which points to the records that need to be retrieved.

5. This cursor can be used in a foreach statement in a synchronous situations by using the `ToEnumerable` adapter method.

```
var cursor = collection.Find(highExamScoreFilter).ToCursor();
foreach (var document in cursor.ToEnumerable())
{
    Console.WriteLine(document);
}
```

This can be accomplished asynchronously with the `ForEachAsync` method:

```
await collection.Find(highExamScoreFilter)
    .ForEachAsync(document => Console.WriteLine(document));
```

## Sorting

With many documents coming back in the result set, it is often helpful to sort the results.

1. We can use the `Sort()` method to accomplish this to see which student had the highest exam score.
   **Note** that the sort method asks the database to do the sorting; it's not performed by the driver or application.

```
// get the highest ID
var sort = Builders<BsonDocument>.Sort.Descending("person_id");
var highestScore = collection.Find(highExamScoreFilter).Sort(sort).First();
Console.WriteLine(highestScore);
```

And we can append the `First()` method to that to just get the top student.

MongoDB Code for sorting by Exam score:

```
db.persons.aggregate([

 // Expand the scores array into a stream of documents
    { $unwind: '$scores' },

    // Filter to 'homework' scores
    { $match: {
        "scores.type": "exam"
    }},

    // Sort in descending order
    { $sort: {
        "scores.score": -1
    }},

    // Merge with itself
    { $lookup:
        {
            from: "persons",
            localField: "_id",
            foreignField: "_id",
            as: "fromItems"
        }
```

```
    },

    // Move nested element (that is an array so I get
    // the first occurrence) to the root
    { $replaceRoot: { newRoot: {$first: "$fromItems" } } }


    ]).pretty()
```

In C#:

```
using MongoDB.Bson;
using MongoDB.Driver;

MongoClient dbClient = new MongoClient("mongodb://localhost:27017");

var database = dbClient.GetDatabase("mongo_sample");
var collection = database.GetCollection<BsonDocument>("persons");

PipelineDefinition<BsonDocument, BsonDocument> pipeline = new BsonDocument[]
{
    new BsonDocument { { "$unwind", "$scores" } },
    new BsonDocument { { "$match", new BsonDocument("scores.score", new BsonDocument {
        { "$gte", 89 }
    }) } },
    new BsonDocument { { "$match", new BsonDocument("scores.type", "exam") } },
    new BsonDocument { { "$sort", new BsonDocument("scores.score", -1) } },
    new BsonDocument { { "$lookup", new BsonDocument
    {
        { "from", "persons"},
        { "localField", "_id"},
        { "foreignField", "_id"},
        { "as", "fromItems"},
    } } },
    new BsonDocument { { "$replaceRoot", new BsonDocument("newRoot", new
BsonDocument("$first", "$fromItems")) } }
};

var result = await collection
    .Aggregate(pipeline)
    .ToListAsync();

foreach (var r in result)
{
    Console.WriteLine(r);
}

//RESULT:
{ "_id" : ObjectId("62be0059b242320fe94a641a"), "person_id" : 10000,
"name" : "Mike", "surname" : "Miller", "scores" : [{ "type" : "exam",
"score" : 91.099999999999994 }, { "type" : "quiz", "score" :
74.900000000000006 }, { "type" : "homework", "score" : 89.900000000000006
}, { "type" : "homework", "score" : 82.099999999999994 }], "class_id" :
480 }
{ "_id" : ObjectId("62be0010492ddc017a345a14"), "person_id" : 10001,
"name" : "Claire", "surname" : "Robinson", "scores" : [{ "type" : "exam",
"score" : 89.5 }, { "type" : "quiz", "score" : 74.900000000000006 }, {
"type" : "homework", "score" : 98.900000000000006 }, { "type" :
"homework", "score" : 82.129310305132194 }], "class_id" : 480 }
```

### Wrap Up

The C# Driver for MongoDB provides many ways to *Read* data from the database and supports both synchronous and asynchronous methods for querying the data. By passing a filter into the `Find()` method, we are able to query for specific records. The syntax to build filters and query the database is straightforward and easy to read, making this step of CRUD operations in C# and MongoDB simple to use.

## Updating Data

To update a document we need two bits to pass into an `Update` command:

- a filter to determine which documents will be updated
- what we update

### Update Filter

1. For our example, we want to filter based on the document with `person_id` equaling 10000.

```
var filter = Builders<BsonDocument>.Filter.Eq("person_id", 10000);
```

### Data to be Changed

2. Next, we want to make the change to the `class_id`. We can do that with `Set()` on the `Update()` method

```
var update = Builders<BsonDocument>.Update.Set("class_id", 483);
```

3. Then we use the `UpdateOne()` method to make the changes

Note here that MongoDB will update at most one document using the `UpdateOne()` method. If no documents match the filter, no documents will be updated.

```
collection.UpdateOne(filter, update);
```

### Array Changes

4. Not all changes are as simple as changing a single field. Let's use a different filter, one that selects a document with a particular score type for quizzes:

```
var arrayFilter = Builders<BsonDocument>.Filter.Eq("person_id", 10000)
                & Builders<BsonDocument>.Filter.Eq("scores.type", "homework");
```

Now if we want to make the change to the quiz score we can do that with `Set()` too, but to identify which particular element should be changed is a little different

We can use the positional `$` operator to access the quiz score in the array

The $ operator on its own says "change the array element that we matched within the query" - the filter matches with scores.type equal to quiz and that element will get updated with the set.

```
var arrayUpdate = Builders<BsonDocument>.Update.Set("scores.$.score", 84.9);
```

And again we use the `UpdateOne()` method to make the changes.

```
collection.UpdateOne(arrayFilter, arrayUpdate);
```

### Additional Update Methods

If you've been reading along in this blog series I've mentioned that the C# driver supports both sync and async interactions with MongoDB. Performing data Updates is no different. There is also an `UpdateOneAsync()` method available. Additionally, for those cases in which multiple documents need to be updated at once, there are `UpdateMany()` or `UpdateManyAsync()` options. The `UpdateMany()` and `UpdateManyAsync()` methods match the documents in the Filter and will update all documents that match the filter requirements.

## Deleting Data

The first step in the deletion process is to create a filter for the document(s) that need to be deleted. In the example for this series, I've been using a document with a person_id value of 10000 to work with. Since I'll only be deleting that single record, I'll use the `DeleteOne()` method (for async situations the `DeleteOneAsync()` method is available). However, when a filter matches more than a single document and all of them need to be deleted, the `DeleteMany()` or `DeleteManyAsync()` method can be used.

I'll define the filter to match the person_id equal to 10000 document:

```
var deleteFilter = Builders<BsonDocument>.Filter.Eq("person_id", 10000);
```

Assuming that we have a collection variable assigned to for the grades collection, we next pass the filter into the DeleteOne() method.

```
collection.DeleteOne(deleteFilter);
```

If that command is run on the `persons` collection, the document with person_id equal to 10000 would be gone. Note here that DeleteOne() will delete the first document in the collection that matches the filter. In our example dataset, since there is only a single student with a person_id equal to 10000, we get the desired results.

For the sake of argument, let's imagine that the rules for the educational institution are incredibly strict. If you get below a score of 60 on the first exam, you are automatically dropped from the course.

We could use a for loop with `DeleteOne()` to loop through the entire collection, find a single document that matches an exam score of less than 60, delete it, and repeat. Recall that `DeleteOne()` only deletes the first document it finds

that matches the filter. While this could work, it isn't very efficient as multiple calls to the database are made. How do we handle situations that require deleting multiple records then? We can use `DeleteMany()`.

### Multiple Deletes

Let's define a new filter to match the exam score being less than 60:

```
var deleteLowExamFilter = Builders<BsonDocument>.Filter.ElemMatch<BsonValue>("scores",
    new BsonDocument {
        { "type", "exam" },
        {"score", new BsonDocument
        {
            { "$lt", 60 }
        }}
    });
```

With the filter defined, we pass it into the DeleteMany() method:

```
collection.DeleteMany(deleteLowExamFilter);
```

With that command being run, all of the student record documents with low exam scores would be deleted from the collection.

## Strongly typed

The option that is strongly typed is the generally preferred way when working with MongoDB in .NET.

The document object is found lower in the object hierarchy. The most general way to represent the document is to use the `BsonDocument` class. The `BsonDocument` is basically a dictionary of keys (strings) and `BsonValues`

1. Create `Person.cs` with `Person` class:

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

namespace MongoDB;

public class Person
{
    [BsonElement("_id")]
    [BsonId]
    [BsonRepresentation(BsonType.ObjectId)]
    public string Id { get; set; }
    [BsonElement("person_id")]
    public int PersonId { get; set; }
    [BsonElement("name")]
    public string Name { get; set; }
    [BsonElement("surname")]
    public string Surname { get; set; }
    [BsonElement("class_id")]
```

```
    public int ClassId { get; set; }
    [BsonElement("scores")]
    public List<ScoreObject> Scores { get; set; }
}

public class ScoreObject
{
    [BsonElement("type")]
    public string ScoreType { get; set; }
    [BsonElement("score")]
    [BsonRepresentation(BsonType.Decimal128)]
    public decimal Score { get; set; }
}
```

2. Insert the first person Lista Grant:

```
using MongoDB;
using MongoDB.Driver;

MongoClient dbClient = new MongoClient("mongodb://localhost:27017");

var database = dbClient.GetDatabase("mongo_sample");
var collection = database.GetCollection<Person>("persons");


var document = new Person
{
    PersonId = 10000,
    Name = "Lisa",
    Surname = "Grant",
    ClassId = 480,
    Scores = new List<ScoreObject>
    {
        new()
        {
            ScoreType = "exam",
            Score = 88.1m
        },
        new()
        {
            ScoreType = "quiz",
            Score = 74.9m
        },
        new()
        {
            ScoreType = "homework",
            Score = 89.9m
        },
        new()
        {
            ScoreType = "homework",
            Score = 82.1m
        }
    }
};

await collection.InsertOneAsync(document);

Console.WriteLine($"Inserted Document {document.Id}");
```

3. Then add
    - Claire Robinson ( personid 10001, exam 98, homework 70)
    - Mike Bell (personid 10002, exam 70, homework 89)

```
using MongoDB;
using MongoDB.Driver;

MongoClient dbClient = new MongoClient("mongodb://localhost:27017");

var database = dbClient.GetDatabase("mongo_sample");
var collection = database.GetCollection<Person>("persons");


var documents = new List<Person>
{
    new Person
    {
        PersonId = 10001,
        Name = "Claire",
        Surname = "Robinson",
        ClassId = 480,
        Scores = new List<ScoreObject>
        {
            new()
            {
                ScoreType = "exam",
                Score = 98.2m
            },
            new()
            {
                ScoreType = "quiz",
                Score = 74.9m
            },
            new()
            {
                ScoreType = "homework",
                Score = 70.9m
            }
        }
    },
    new Person
    {
        PersonId = 10002,
        Name = "Mike",
        Surname = "Bell",
        ClassId = 480,
        Scores = new List<ScoreObject>
        {
            new()
            {
                ScoreType = "exam",
                Score = 70.6m
            },
            new()
            {
                ScoreType = "quiz",
                Score = 74.9m
            },
            new()
```

```
            {
                ScoreType = "homework",
                Score = 89.9m
            }
        }
    }
};


await collection.InsertManyAsync(documents);

Console.WriteLine($"Inserted Documents:");
foreach (var d in documents)
{
    Console.WriteLine($"{d.Id}");

}
```

4. Then add filter for Name equal to “Mike”

```
using System.Text.Json;
using MongoDB;
using MongoDB.Driver;

MongoClient dbClient = new MongoClient("mongodb://localhost:27017");

var database = dbClient.GetDatabase("mongo_sample");
var collection = database.GetCollection<Person>("persons");

var filter = Builders<Person>.Filter.Eq(x => x.Name, "Mike");
var res = await collection.Find(filter).ToListAsync();

foreach (var item in res)
{
    Console.WriteLine($"{item.Id}: {item.Name},
{JsonSerializer.Serialize(item.Scores)}");
}
```

5. List all elements sorting by name (two ways: with and without filterdefinition)

```
using System.Text.Json;
using MongoDB;
using MongoDB.Driver;

MongoClient dbClient = new MongoClient("mongodb://localhost:27017");

var database = dbClient.GetDatabase("mongo_sample");
var collection = database.GetCollection<Person>("persons");

var filter = Builders<Person>.Filter.Empty;

var sort = Builders<Person>.Sort.Ascending("name");

var filterDefinitionResult = await collection.Find(filter).Sort(sort).ToListAsync();

var result = await collection.Find(_ => true).Sort(sort).ToListAsync();
```

```
Console.WriteLine("With Filter definition");
foreach (var item in filterDefinitionResult)
{
    Console.WriteLine($"{item.Id}: {item.Name}, {JsonSerializer.Serialize(item.Scores)}");
}

Console.WriteLine("--");
Console.WriteLine("Without Filter definition");
foreach (var item in result)
{
    Console.WriteLine($"{item.Id}: {item.Name}, {JsonSerializer.Serialize(item.Scores)}");
}
```

6. Strongly typed query to retrieve who have exam>85, ordered by descending exam score

```
PipelineDefinition<Person, Person> pipeline = new BsonDocument[]
{
    new BsonDocument { { "$unwind", "$scores" } },
    new BsonDocument { { "$match", new BsonDocument("scores.score", new BsonDocument {
        { "$gte", 85 }
    }) } },
    new BsonDocument { { "$match", new BsonDocument("scores.type", "exam") } },
    new BsonDocument { { "$sort", new BsonDocument("scores.score", -1) } },
    new BsonDocument { { "$lookup", new BsonDocument
    {
        { "from", "persons"},
        { "localField", "_id"},
        { "foreignField", "_id"},
        { "as", "fromItems"},
    } } },
    new BsonDocument { { "$replaceRoot", new BsonDocument("newRoot", new
BsonDocument("$first", "$fromItems")) } }
};

var result = await collection
    .Aggregate(pipeline)
    .ToListAsync();

foreach (var r in result)
{
    Console.WriteLine(r.Name);
    Console.WriteLine(JsonSerializer.Serialize(r.Scores));
}
```

a. First step: first simple query

```
var result2 = collection
    .AsQueryable();

foreach (var r in result2)
{
    Console.WriteLine(JsonSerializer.Serialize(r));
}
```

b. Second step: explode the array

```
var result2 = collection
```

```
    .AsQueryable()
    .SelectMany(p => p.Scores, (p, s) => new
    {
        p.Id,
        s.ScoreType,
        s.Score,
    });

foreach (var r in result2)
{
    Console.WriteLine(JsonSerializer.Serialize(r));
}
```

c. Add filtering and sorting

```
var result3 = collection
    .AsQueryable()
    .SelectMany(p => p.Scores, (p, s) => new
    {
        p.Id,
        s.ScoreType,
        s.Score,
    })
    .Where(s => s.Score >=85 && s.ScoreType == "exam")
    .OrderByDescending(x => x.Score);
foreach (var r in result3)
{
    Console.WriteLine(JsonSerializer.Serialize(r));
}
```

d. Join with the original collection to access all members and add `ToList()` to execute the query

```
var result3 = collection
    .AsQueryable()
    .SelectMany(p => p.Scores, (p, s) => new
    {
        p.Id,
        s.ScoreType,
        s.Score,
    })
    .Where(s => s.Score >= 85 && s.ScoreType == "exam")
    .OrderByDescending(x => x.Score)
    .Join(collection.AsQueryable(),
        x => x.Id,
        p => p.Id,
        (x, p) => p
        )
    .ToList();

foreach (var r in result3)
{
    Console.WriteLine(JsonSerializer.Serialize(r));
}
```

e. Here is the complete code

```
using System.Text.Json;
using MongoDB;
using MongoDB.Bson;
using MongoDB.Driver;

MongoClient dbClient = new MongoClient("mongodb://localhost:27017");

var database = dbClient.GetDatabase("mongo_sample");
var collection = database.GetCollection<Person>("persons");

PipelineDefinition<Person, Person> pipeline = new BsonDocument[]
{
    new BsonDocument { { "$unwind", "$scores" } },
    new BsonDocument { { "$match", new BsonDocument("scores.score", new BsonDocument {
        { "$gte", 85 }
    }) } },
    new BsonDocument { { "$match", new BsonDocument("scores.type", "exam") } },
    new BsonDocument { { "$sort", new BsonDocument("scores.score", -1) } },
    new BsonDocument { { "$lookup", new BsonDocument
    {
        { "from", "persons"},
        { "localField", "_id"},
        { "foreignField", "_id"},
        { "as", "fromItems"},
    } } },
    new BsonDocument { { "$replaceRoot", new BsonDocument("newRoot", new
BsonDocument("$first", "$fromItems")) } }
};

var result = await collection
    .Aggregate(pipeline)
    .ToListAsync();

foreach (var r in result)
{
    Console.WriteLine(r.Name);
    Console.WriteLine(JsonSerializer.Serialize(r.Scores));
}

var result2 = collection
    .AsQueryable()
    .SelectMany(p => p.Scores, (p, s) => new
    {
        p.Id,
        s.ScoreType,
        s.Score,
    })
    .Where(s => s.Score >= 85 && s.ScoreType == "exam")
    .OrderByDescending(x => x.Score)
    .Join(collection.AsQueryable(),
        x => x.Id,
        p => p.Id,
        (x, p) => p
    )
    .ToList();


Console.WriteLine();
Console.WriteLine();
Console.WriteLine();
```

```
foreach (var r in result2)
{
    Console.WriteLine(r.Name);
    Console.WriteLine(JsonSerializer.Serialize(r.Scores));
}
```

## Another way to write it

7. Create PersonUnwindResult.cs file with PersonUnwindResult class:

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

namespace MongoDB;

public class PersonUnwindResult
{
    [BsonId, BsonElement("_id"), BsonRepresentation(BsonType.ObjectId)]
    public string Id { get; set; }
    [BsonElement("person_id")]
    public int PersonId { get; set; }
    [BsonElement("name")]
    public string Name { get; set; }
    [BsonElement("surname")]
    public string Surname { get; set; }
    [BsonElement("class_id")]
    public int ClassId { get; set; }
    [BsonElement("scores")]
    public ScoreObject Scores { get; set; }
}
```

8. In Program.cs now add this:

```
var filter =
    Builders<PersonUnwindResult>.Filter.And(
        Builders<PersonUnwindResult>.Filter.Eq(x => x.Scores.ScoreType, "exam"),
        Builders<PersonUnwindResult>.Filter.Gte(x => x.Scores.Score, 85));


var result3 = collection
    .Aggregate()
    .Unwind<Person, PersonUnwindResult>(x => x.Scores)
    //.Match(x => x.Scores.ScoreType =="exam" && x.Scores.Score > 85)
    .Match(filter)
    .SortByDescending(x => x.Scores.Score)
    .ToList();

foreach (var r in result3)
{
    Console.WriteLine(r.Name);
    Console.WriteLine(JsonSerializer.Serialize(r));
}
```

9. The complete code will now look like

```
using System.Text.Json;
using MongoDB;
using MongoDB.Bson;
using MongoDB.Driver;

MongoClient dbClient = new MongoClient("mongodb://localhost:27017");

var database = dbClient.GetDatabase("mongo_sample");
var collection = database.GetCollection<Person>("persons");

PipelineDefinition<Person, Person> pipeline = new BsonDocument[]
{
    new BsonDocument { { "$unwind", "$scores" } },
    new BsonDocument { { "$match", new BsonDocument("scores.score", new BsonDocument {
        { "$gte", 85 }
    }) } },
    new BsonDocument { { "$match", new BsonDocument("scores.type", "exam") } },
    new BsonDocument { { "$sort", new BsonDocument("scores.score", -1) } },
    new BsonDocument { { "$lookup", new BsonDocument
    {
        { "from", "persons"},
        { "localField", "_id"},
        { "foreignField", "_id"},
        { "as", "fromItems"},
    } } },
    new BsonDocument { { "$replaceRoot", new BsonDocument("newRoot", new
BsonDocument("$first", "$fromItems")) } }
};

var result = await collection
    .Aggregate(pipeline)
    .ToListAsync();

foreach (var r in result)
{
    Console.WriteLine(r.Name);
    Console.WriteLine(JsonSerializer.Serialize(r.Scores));
}

var result2 = collection
    .AsQueryable()
    .SelectMany(p => p.Scores, (p, s) => new
    {
        p.Id,
        s.ScoreType,
        s.Score,
    })
    .Where(s => s.Score >= 85 && s.ScoreType == "exam")
    .OrderByDescending(x => x.Score)
    .Join(collection.AsQueryable(),
        x => x.Id,
        p => p.Id,
        (x, p) => p
    )
    .ToList();


Console.WriteLine();
Console.WriteLine();
Console.WriteLine();

foreach (var r in result2)
```

```
{
    Console.WriteLine(r.Name);
    Console.WriteLine(JsonSerializer.Serialize(r.Scores));
}


Console.WriteLine();
Console.WriteLine();
Console.WriteLine();

var filter =
    Builders<PersonUnwindResult>.Filter.And(
        Builders<PersonUnwindResult>.Filter.Eq(x => x.Scores.ScoreType, "exam"),
        Builders<PersonUnwindResult>.Filter.Gte(x => x.Scores.Score, 85));


var result3 = collection
    .Aggregate()
    .Unwind<Person, PersonUnwindResult>(x => x.Scores)
    //.Match(x => x.Scores.ScoreType =="exam" && x.Scores.Score > 80)
    .Match(filter)
    .SortByDescending(x => x.Scores.Score)
    .ToList();

foreach (var r in result3)
{
    Console.WriteLine(r.Name);
    Console.WriteLine(JsonSerializer.Serialize(r));
}
```