.NET Core Rest API:

We will have a MongoDB service which is going to be responsible for connecting and interacting to MongoDB.

We will have a Controller for each of the endpoints. The endpoint functions in that controller will communicate to the MongoDB service.

We will have various configuration files and classes to represent our models for working with our data.

In MongoDB:

"Load Sample Dataset" in my cluster.

We will use the sample_mflix database but will create our custom playlist collection.

Database Access section has user details.

Playlist Model:

1.

[BsonId]

- This attribute designates the Id property as the unique identifier for each Playlist object within your MongoDB collection.
- MongoDB will automatically generate a unique ObjectId value for this property if you don't provide one when creating a new Playlist instance.

[BsonRepresentation(BsonType.ObjectId)]

- This attribute specifies how the `Id` property should be represented when stored in MongoDB.
- `BsonType.ObjectId` tells MongoDB to store the `Id` as a BSON ObjectId, which is MongoDB's default and recommended way to handle unique identifiers.
- BSON ObjectIds are 12-byte values that ensure uniqueness and can also encode some timestamp information.

public string? Id { get; set; }

```
    [BsonId]
    [BsonRepresentation(BsonType.ObjectId)]
    public string? Id { get; set; }
```

2.

```
    [BsonElement("items")]
    [JsonPropertyName("items")]
    public List<string> movieIds { get; set; } = null!;
```

- `[BsonElement("items")]`
  This attribute, from the `MongoDB.Bson.Serialization.Attributes` namespace, tells the MongoDB driver that when serializing (saving) or deserializing (loading) this `Playlist` object to/from the database, the `movieIds` property should be represented by the field named "items" in the MongoDB document.
  In simpler terms, it maps the C# property name `movieIds` to the MongoDB field name **`items`**.
- `[JsonPropertyName("items")]`
  This attribute, from the `System.Text.Json.Serialization` namespace, serves a similar purpose but in a different context. It instructs the `System.Text.Json` serializer that when converting this `Playlist` object to or from JSON format, the `movieIds` property should be represented by the JSON property named "items".

Singleton:

```
// import MongoDB dependencies
using MongoDB.Models;
using MongoDB.Services;

var builder = WebApplication.CreateBuilder(args);

// bind our settings with the MongoDB service
builder.Services.Configure<MongoDBSettings>(builder.Configuration.GetSection("Mon
goDB"));
// create the singleton instance
builder.Services.AddSingleton<MongoDBService>();
// When we start our applicaiton, we are creating the MongoDB Singleton service
and are able to use that within other files
```

In this context, singleton means that there will only be one instance of the MongoDBService created and shared throughout your entire application.

Here's what that implies:

Single Point of Access: Any part of your application that needs to interact with MongoDB will use this same MongoDBService instance.

Shared State: If the MongoDBService maintains any internal state (like a connection to the database, cached data, etc.), that state will be shared across all parts of the application that use this service.

Lifecycle: The MongoDBService instance will be created when your application starts and will typically live for the entire duration of your application's runtime.

Important Considerations:

- **Thread Safety:** If your `MongoDBService` is accessed concurrently from multiple threads, you need to ensure it's designed to be thread-safe to prevent any data corruption or unexpected behavior.
- **Testing:** Singletons can sometimes make unit testing a bit trickier because they introduce a global dependency. Consider using techniques like dependency injection or mocking to make your code more testable.

Next step:

1. Add functions to the MongoDB service to interact with MongoDB.
2. Create a controller which has endpoint functions that communicate with that service.

PlaylistController:

1.

A controller is an instantiable class, usually public, in which at least one of the following conditions is true:
The class name is suffixed with Controller.
The class inherits from a class whose name is suffixed with Controller.
The [Controller] attribute is applied to the class.

2.
The methods in the PlaylistController return a **Task** because they are marked as **async**:

- **Asynchronous Operations:** The `async` keyword indicates that the method contains asynchronous operations, likely interacting with the database or external services. These operations can take time to complete, and blocking the current thread while waiting would be inefficient.
- **Task Represents the Ongoing Work:** The `Task` returned by an async method represents the ongoing asynchronous operation. It allows the calling code to continue executing without waiting for the operation to finish.
- **Awaiting the Task:** When the calling code needs the result of the asynchronous operation, it can use the `await` keyword on the `Task`. This will pause the execution of the calling code until the asynchronous operation completes and the result is available.

Benefits of using async/await:

- **Improved Scalability:** Async/await enables your application to handle more concurrent requests efficiently, as threads are not blocked waiting for I/O-bound operations to complete.

3.

[HttpPost]

public async Task<IActionResult> Post([FromBody] Playlist playlist) {}

- `async Task<IActionResult>`:
  The method is asynchronous and returns a `Task` that will eventually complete with an `IActionResult`.

  `IActionResult` allows the method to return various types of HTTP responses (e.g., success, error, redirect) depending on the outcome of the operation.

- `[FromBody] Playlist playlist`:
  `[FromBody]`: This attribute instructs ASP.NET Core to bind the incoming request's body (typically JSON data) to a `Playlist` object.

`Playlist playlist`: The deserialized `Playlist` object will be available within the method for further processing (e.g., saving to a database).

How it Works:

Client Sends Request: A client (e.g., web browser, mobile app) sends an HTTP POST request to the appropriate URL. The URL is determined by the **`[Route]`** attribute on the controller.

Request Body: The request includes a JSON representation of the new playlist in the body.

ASP.NET Core Routing: ASP.NET Core's routing mechanism identifies the `PlaylistController` and the `Post` method as the handler for this request based on the URL and the `[HttpPost]` attribute.

Model Binding: ASP.NET Core deserializes the JSON data in the request body into a `Playlist` object and passes it as an argument to the `Post` method.

Method Execution: The `Post` method executes, likely performing actions like:

- Validating the `Playlist` data
- Saving the new playlist to the database using the `_mongoDBService`

Response: The method returns an `IActionResult`. This could be:

- `Ok()` - indicating success, possibly with the newly created playlist's ID in the response body
- `BadRequest()` - if the playlist data is invalid
- Other status codes depending on the specific outcome

How to run the program:

We are using Swagger - we do have our endpoints created under the API path.
Use: dotnet run
Go to: http://localhost:5074/swagger/

Making the Playlist collection:

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

// _
// Null-forgiving Operator (!)
// changed the mapping - The document will have a user field but the local class
will be used as username
public class Playlist {
    public ObjectId _id { get; set; }
    [BsonElement("user")]
    public string username { get; set; } = null!; // nullable
    public List<string> items { get; set; } = null!; // nullable

    // constructor
    public Playlist(string username, List<string> movieIds) {
        this.username = username;
        this.items = movieIds;
    }

}
```

MongoDB and C# Integration:
ObjectId _id: This is a standard property for MongoDB documents, representing the **unique identifier**.
The [BsonElement("user")] attribute specifies that the username property in your class will be stored as user in the MongoDB document.

Null-forgiving Operator (null!):
The null! operator is used to suppress nullable warnings in C#. Even though username and items are being initialized with null, the operator tells the compiler that you're aware and will handle initialization later, which you're doing in the constructor.

Constructor:

The constructor takes username and movieIds as parameters and assigns them to the respective properties.

This ensures that by the time an instance of Playlist is created, both username and items are non-null, fulfilling the contract expected by non-nullable reference types.

In This Context:

Non-nullable Reference Types Contract: In C# (especially with C# 8.0 and later), when you declare a property as a non-nullable reference type (e.g., string, List<string>), the compiler expects that these properties will never hold a null value once they are initialized. This expectation is the "contract" you have with the compiler and other developers who use your code.