# Building CRUD Endpoints that Interact with MongoDB Using .NET Core

When building CRUD endpoints for this project, we'll need to bounce between two different locations within our project. We'll need to define the endpoint within a controller and do the work within our service.

Create "Controllers/PlaylistController.cs" and add the following code:

```
using Microsoft.AspNetCore.Mvc;
using MongoDB.Services;
using MongoDB.Models;

namespace MongoDB.Controllers;
// We will have to define the endpoint within the controller and do the work
within our service.

// The class PlaylistController extends Controller

// Path for accessing the different endpoint functions for this particular
controller class
[ApiController]
[Route("api/[controller]")]
public class PlaylistController: Controller {
    private readonly MongoDBService _mongoDBService;
    // private readonly variable for the MongoDBService to access it locally
within this particular controller class

    // constructor method - we will pass in that mongoDBService which will be
handled for us
    public PlaylistController(MongoDBService mongoDBService) {
        _mongoDBService = mongoDBService;
    }

    // defining the endpoints

    // returns a Task because it's async
    [HttpGet]
    public async Task<List<Playlist>> Get() {
        return await _mongoDBService.GetAsync();
    }
```

```
    [HttpPost]
    public async Task<IActionResult> Post([FromBody] Playlist playlist) {
        await _mongoDBService.CreateAsync(playlist);
        return CreatedAtAction(nameof(Get), new { id = playlist.Id } , playlist);
    }

    // Put is common for update operations
    // We want to find which document we want to update
    // options - in the body, query parameter, route parameter, etc.

    // Add a new item to the existing playlist
    // Specify the id passed in from the user - from that route parameter
    // Accept a payload from the body - that's going to contain the actual
movieId

    // route parameter
    [HttpPut("{id}")]
    public async Task<IActionResult> AddToPlaylist(string id, [FromBody] string
movieId) {
        await _mongoDBService.AddToPlaylistAsync(id, movieId);
        return NoContent();
    }

    [HttpDelete("{id}")]
    public async Task<IActionResult> Delete(string id) {
        await _mongoDBService.DeleteAsync(id);
        return NoContent();
    }

}
```

In the above `PlaylistController` class, we have a constructor method that gains access to our singleton service class. Then we have a series of endpoints for this particular controller. We could add far more endpoints than this to our controller, but it's not necessary for this example.

Let's start with creating data through the POST endpoint. To do this, it's best to start in the "Services/MongoDBService.cs" file:

```
    public async Task CreateAsync(Playlist playlist) {
        await _playlistCollection.InsertOneAsync(playlist);
        return;
    }
```

We had set the `_playlistCollection` in the constructor method of the service, so we can now use the `InsertOneAsync` method, taking a passed `Playlist` variable and inserting it. Jumping back into the "Controllers/PlaylistController.cs," we can add the following:

```
    [HttpPost]
    public async Task<IActionResult> Post([FromBody] Playlist playlist) {
        await _mongoDBService.CreateAsync(playlist);
        return CreatedAtAction(nameof(Get), new { id = playlist.Id } , playlist);
    }
```

What we're saying is that when the endpoint is executed, we take the `Playlist` object from the request, something that .NET Core parses for us, and pass it to the `CreateAsync` function that we saw in the service. After the insert, we return some information about the interaction.

It's important to note that in this example project, we won't be validating any data flowing from HTTP requests.

Let's jump to the read operations.

Head back into the "Services/MongoDBService.cs" file and add the following function:

```
    public async Task<List<Playlist>> GetAsync() {
```

```
        // BsonDocument - every single document of this collection is going to
come back
        // ToListAsync - we don't want to work with a particular cursor for this
example
        return await _playlistCollection.Find(new BsonDocument()).ToListAsync();
    }
```

The above `Find` operation will return all documents that exist in the collection. If you wanted to, you could make use of the `FindOne` or provide filter criteria to return only the data that you want. We'll explore filters shortly.

With the service function ready, add the following endpoint to the "Controllers/PlaylistController.cs" file:

```
    [HttpGet]
    public async Task<List<Playlist>> Get() {
        return await _mongoDBService.GetAsync();
    }
```

The next CRUD stage to take care of is the updating of data. Within the "Services/MongoDBService.cs" file, add the following function:

```
    public async Task AddToPlaylistAsync(string id, string movieId){
        FilterDefinition<Playlist> filter = Builders<Playlist>.Filter.Eq("id",
id);
        UpdateDefinition<Playlist> update =
Builders<Playlist>.Update.AddToSet<string>("movieId", movieId);
        await _playlistCollection.UpdateOneAsync(filter, update);
        return;
    }
```

Rather than making changes to the entire document, we're planning on adding an item to our playlist and nothing more. To do this, we set up a match filter to determine which document or documents should receive the update. In this case, we're matching on the id which is going to be unique. Next, we're defining the update criteria, which is an `AddToSet` operation that will only add an item to the array if it doesn't already exist in the array.

The UpdateOneAsync method will only update one document even if the match filter returned more than one match.

In the "Controllers/PlaylistController.cs" file, add the following endpoint to pair with the AddToPlayListAsync function:

```
    [HttpPut("{id}")]
    public async Task<IActionResult> AddToPlaylist(string id, [FromBody] string
movieId) {
        await _mongoDBService.AddToPlaylistAsync(id, movieId);
        return NoContent();
    }
```

In the above PUT endpoint, we are taking the `id` from the route parameters and the `movieId` from the request body and using them with the `AddToPlaylistAsync` function.

This brings us to our final part of the CRUD spectrum. We're going to handle deleting of data.

In the "Services/MongoDBService.cs" file, add the following function:

```
    public async Task DeleteAsync(string id) {
        FilterDefinition<Playlist> filter = Builders<Playlist>.Filter.Eq("id",
id);
        await _playlistCollection.DeleteOneAsync(filter);
        return;

    }
```

The above function will delete a single document based on the filter criteria.

The filter criteria, in this circumstance, is a match on the id which is always

going to be unique. Your filters could be more extravagant if you wanted.

To bring it to an end, the endpoint for this function would look like the following in the "Controllers/PlaylistController.cs" file:

```
    [HttpDelete("{id}")]
    public async Task<IActionResult> Delete(string id) {
        await _mongoDBService.DeleteAsync(id);
        return NoContent();
    }
```

We only created four endpoints, but you could take everything we did and create 100 more if you wanted to.