Implementing a MongoDB repository

In this lesson we will learn about the repository pattern and how to implement a repository to manage the Catalog service items in a MongoDB database.

What is a repository?

A repository is an abstraction between the data layer and the business layer of an application.

To understand why you may want to use a repository, let's say that today we have the application logic of our service talking directly to our MongoDB database. However, a few months or years from now, requirements change and we need to switch to a different database provider. Given the way things have been setup we would likely need o rewrite a good chunk of our application logic in order to talk to the new database, which is pretty bad.

Instead what we can do is introduce a repository. This repository is the only one that knows how to talk, initially, to our MongoDB database and is the only one that our application logic will interface with. Then, if we ever get the requirement to move to another database the only thing that we would need to change is our repository but the application logic stays the same.

So the repository pattern is important because it decouples the application logic from the data layer and minimizes duplicate data access logic across our service.

Using MongoDB to store items

We will be using MongoDB as our database of choice for all the microservices in this course.

MongoDB is a document-oriented **NoSQL database which stores data in JSON-like documents with dynamic schema**

We will prefer a NoSQL solution (as opposed to a relational database) for our microservices because:

- We won't need relationships across the data, because each microservice manages its own database
- We don't need ACID guarantees, where ACID stands for atomicity, consistency, isolation and durability, which are properties of database transactions that we won't need in our services.
- We won't need to write complex queries, since most of our service queries will be able to find everything they need in a single document type
- Need low latency, high availability and high scalability, which are classic features of NoSQL databases

Let's go ahead and implement our ItemsRepository.

Implementing the repository

1. Before implementing our repository, we will need to decide which his going to be the class that represents the objects that will be managed by the repository and that will make their way to our database. The class we use for this should not be confused with our DTOs because we want to have the freedom of updating how we store the items in the database at any given point, regardless of the contract that we need to honor with our service clients.

   We will then give the name of “Entities” to the classes that our repository use.

2. Add **Entities** directory

3. Add Item.cs:

```
public class Item
{
    public Guid Id { get; set; }

    public string Name { get; set; }

    public string Description { get; set; }

    public decimal Price { get; set; }

    public DateTimeOffset CreatedDate { get; set; }
}
```

4. dotnet add package MongoDB.Driver

5. Add Repositories folder

6. Add ItemsRepository to Repositories folder:

```
public class ItemsRepository
{
    private const string collectionName = "items";
    private readonly IMongoCollection<Item> dbCollection;
    private readonly FilterDefinitionBuilder<Item> filterBuilder = Builders<Item>.Filter;

    public ItemsRepository()
    {
        var mongoClient = new MongoClient($"mongodb://localhost:27017");
        var database = mongoClient.GetDatabase("Catalog");
        dbCollection = database.GetCollection<Item>(collectionName);
    }

    // Now, for our repository public methods we will be using the Asynchronous programming
    model. This will avoid performance bottlenecks and enhance the overall responsiveness of our
    service.

    // To take advantage of this model all of our methods will become asynchronous by returning
    async Task and by using the await keyword any time they interact with the database. We will
    also use the Async suffix on all the methods to surface the fact that the methods are
    asynchronous.

    public async Task<IReadOnlyCollection<Item>> GetAllAsync()
    {
        return await dbCollection.Find(filterBuilder.Empty).ToListAsync();
    }

    public async Task<Item> GetAsync(Guid id)
    {
        FilterDefinition<Item> filter = filterBuilder.Eq(entity => entity.Id, id);
        return await dbCollection.Find<Item>(filter).FirstOrDefaultAsync();
    }

    public async Task CreateAsync(Item entity)
    {
        if (entity == null)
        {
            throw new ArgumentNullException(nameof(entity));
        }

        await dbCollection.InsertOneAsync(entity);
    }

    public async Task UpdateAsync(Item entity)
    {
        if (entity == null)
        {
            throw new ArgumentNullException(nameof(entity));
        }

        FilterDefinition<Item> filter = filterBuilder.Eq(existingEntity => existingEntity.Id, entity.Id);
        await dbCollection.ReplaceOneAsync(filter, entity);
    }
    public async Task RemoveAsync(Guid id)
    {
        FilterDefinition<Item> filter = filterBuilder.Eq(entity => entity.Id, id);
        await dbCollection.DeleteOneAsync(filter);
    }
}
```

7. Add Extensions.cs:

```
public static class Extensions
{
    public static ItemDto AsDto(this Item item)
    {
        return new ItemDto(item.Id, item.Name, item.Description, item.Price, item.CreatedDate);
    }
}
```

8. Update ItemsController:

```
[ApiController]
[Route("items")]
public class ItemsController : ControllerBase
{
    private readonly ItemsRepository itemsRepository = new();

    [HttpGet]
    public async Task<ActionResult<IEnumerable<ItemDto>>> GetAsync()
    {
        var items = (await itemsRepository.GetAllAsync())
                    .Select(item => item.AsDto());

        return Ok(items);
    }

    [HttpGet("{id}")]
    public async Task<ActionResult<ItemDto>> GetByIdAsync(Guid id)
    {
        var item = await itemsRepository.GetAsync(id);

        if (item == null)
        {
            return NotFound();
        }
    return item.AsDto();
}

[HttpPost]
public async Task<ActionResult<ItemDto>> PostAsync(CreateItemDto createItemDto)
{
    var item = new Item
    {
        Name = createItemDto.Name,
        Description = createItemDto.Description,
        Price = createItemDto.Price,
        CreatedDate = DateTimeOffset.UtcNow
    };

    await itemsRepository.CreateAsync(item);

    return CreatedAtAction(nameof(GetByIdAsync), new { id = item.Id }, item.AsDto());
}

[HttpPut("{id}")]
public async Task<IActionResult> PutAsync(Guid id, UpdateItemDto updateItemDto)
{
    var existingItem = await itemsRepository.GetAsync(id);

    if (existingItem == null)
    {
        return NotFound();
    }

    existingItem.Name = updateItemDto.Name;
    existingItem.Description = updateItemDto.Description;
    existingItem.Price = updateItemDto.Price;

    await itemsRepository.UpdateAsync(existingItem);

    return NoContent();
}

[HttpDelete("{id}")]
public async Task<IActionResult> DeleteAsync(Guid id)
{
    var item = await itemsRepository.GetAsync(id);
```

```
        if (item == null)
        {
            return NotFound();
        }

        await itemsRepository.RemoveAsync(item.Id);

        return NoContent();
    }
}
```

9. Our REST API is ready to use our new ItemsRepository to store and query items in a MongoDB database. However, we still don’t have a MongoDB database, or even a MongoDB server. That’s a problem we can easily solve with Docker.

In the next lesson we will start using Docker to run the infrastructure components needed by our microservices.