

ASP.NET Web APIs

ASP.NET Core Web API

המשך הפרוייקט משיעור קודם:

מסמך אפיון של API לכרטיסיות:

כרטיסיות:

<https://documenter.getpostman.com/view/25008645/2s9YXcd5BE>

משתמשים

<https://documenter.getpostman.com/view/25008645/2s9YXcd5BL>

ASP.NET Core Web API

Intro to Data transfer objects

הלקוח צריך לשלוח בבקשה
רק את השדות הבאים

Title
Description

בדטהבייס נשמור יותר שדות

Id
Title
Description
CreatedAt
UpdatedAt
Likes

ASP.NET Core Web API

מודל לכתובת: כולל ולידציות:

```
public class Address
{
    [MinLength(2), MaxLength(256)]
    public string? State { get; set; }
    [MinLength(2), MaxLength(256)]
    [Required]
    public required string Country { get; set; }

    [MinLength(2), MaxLength(256)]
    [Required]
    public required string City { get; set; }

    [MinLength(2), MaxLength(256)]
    [Required]
    public required string Street { get; set; }

    [Required]
    [Range(1, 1000)]
    public int HouseNumber { get; set; }

    [Required]
    [Range(1, 1000)]
    public int Zip { get; set; } = 0;
}
```

ASP.NET Core Web API

מודל לתמונה:

```
public class Image
{
    [Required, Url]
    public required string Url { get; set; }
    [Required]
    [StringLength(256, MinimumLength = 2,
        ErrorMessage = "Must provide a valid alt between 2 - 256 characters")]
    public required string Alt { get; set; }
}
```

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

    [Required, StringLength(maximumLength: 256, MinimumLength = 2)]
    public required string Title { get; set; }

    [Required, StringLength(maximumLength: 256, MinimumLength = 2)]
    public required string Subtitle { get; set; }

    [Required, StringLength(1024)]
    public required string Description { get; set; }

    [Required, Phone]
    public required string Phone { get; set; }

    [Required, EmailAddress]
    public required string Email { get; set; }

    [Url]
    public string? Web { get; set; }

    public Image? Image { get; set; }
    [Required]
    public required Address Address { get; set; }

    [Required]
    public long BizNumber { get; set; }

    public List<string> Likes { get; set; } = [];

    public DateTime CreateAt { get; set; } = DateTime.UtcNow;

    [JsonPropertyName("user_id")]
    public required string User_Id { get; set; }
}
```

מודל לכרטיסיה כולל Id וגם
לייקים ומידע נוסף שלא מגיע מהשתמש:

כדי לקבל מידע שמגיע מהמשתמש
ניצור DTO (בעמוד הבא)

```

public class CardCreateDto
{

    [Required, StringLength(maximumLength: 256, MinimumLength = 2)]
    public required string Title { get; set; }

    [Required, StringLength(maximumLength: 256, MinimumLength = 2)]
    public required string Subtitle { get; set; }

    [Required, StringLength(1024)]
    public required string Description { get; set; }

    [Required, Phone]
    public required string Phone { get; set; }

    [Required, EmailAddress]
    public required string Email { get; set; }

    [Url]
    public string? Web { get; set; }

    public Image? Image { get; set; }

    [Required]
    public required Address Address { get; set; }

}

```

אובייקט של DTO

DTO = Data transfer object

והרבה פעמים המידע שאנו שומרים
בדטה-בייס שונה בחלקו מהמידע שמגיע מהלקוח.

העניין הוא שעכשיו יש לנו 2 אובייקטים: CardDto=>Card

ואנחנו צריכים את שניהם –
את האובייקט ה-1: כדי לייצג את המידע שנכנס.
את האובייקט ה-2: כדי לייצג את המידע השמור בדטהבייס.

*כדי לשמור אובייקט מסוג Card
נצטרך להפוך את CardDto לקיבלנו מהמשתמש לCard
ששומרים בדטהבייס (בעמוד הבא)

```
public static class CardCreateDtoExtensions
{
    public static Models.Card ToCard(this CardCreateDto cardCreateDto, string userId)
    {
        return new Models.Card
        {
            Title = cardCreateDto.Title,
            Subtitle = cardCreateDto.Subtitle,
            Description = cardCreateDto.Description,
            Phone = cardCreateDto.Phone,
            Email = cardCreateDto.Email,
            Web = cardCreateDto.Web,
            Image = cardCreateDto.Image,
            Address = cardCreateDto.Address,
            User_Id = userId
        };
    }
}
```

המרה בעזרת מתודת הרחבה.

אפשר להשתמש ב Extensions גם בשביל להרחיב מחלקות
שלא אנחנו יצרנו.

```
using Humanizer;  
using MongoDB.Driver;
```

```
namespace RZB.Data
```

```
{
```

```
    public static class IMongoDatabaseExtensions
```

```
    {
```

```
        //database extensions:
```

```
        public static IMongoCollection<T> GetCollection<T>(this IMongoDatabase database)
```

```
        {
```

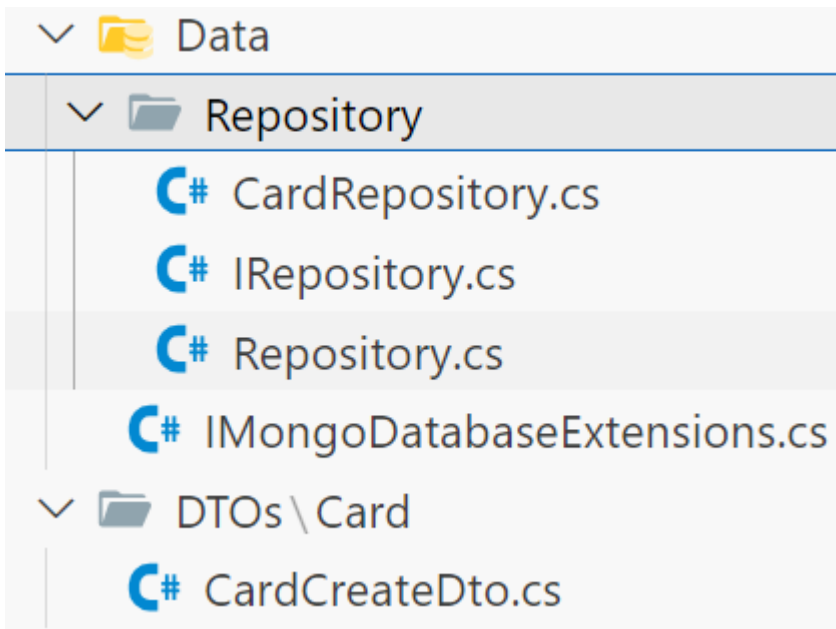
```
            return database.GetCollection<T>(typeof(T).Name.Pluralize());
```

```
        }
```

```
    }
```

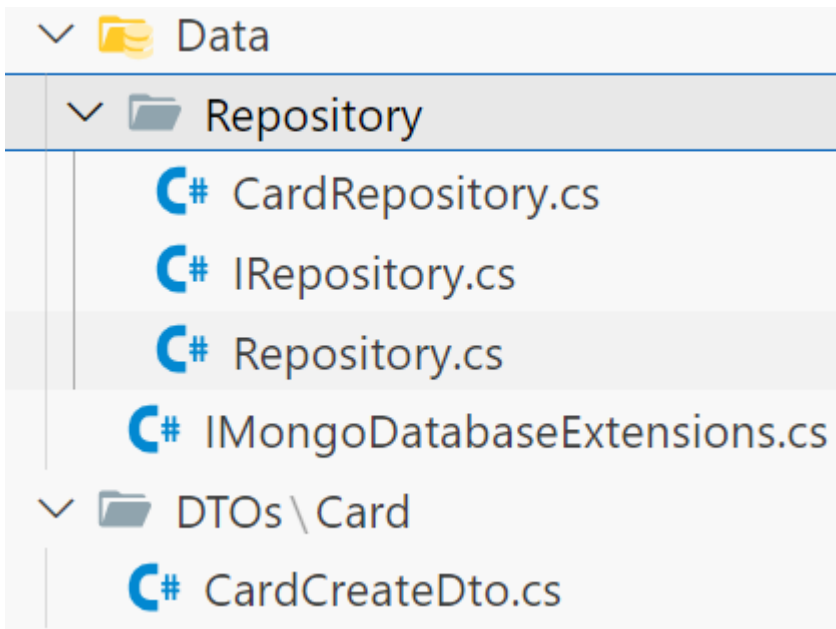
```
}
```

ספריית Nuget שעוסקת בהמרה ללשון
רבים ועוד יכולות כגון הצגה אנושית
של זמנים ותאריכים.



עכשיו כדי לעבוד עם כרטיסיות צריך ליצור Repository חדש... העניין הוא שהקוד יהיה בדיוק אותו דבר כמו הקוד הקודם שלנו עם שינוי אחד – במקום Movie יהיה לנו עכשיו Card או User.

במילים אחרות – נרצה להפוך את ה Repository שלנו לגנרי וכך נכתוב פעם אחת Repository ונוכל להשתמש בו הרבה פעמים.



ממשק גנרי
(נוסיף לו אילוץ בהמשך)
(בשקפים הבאים)

```
public interface IRepository<T>
{
    2 references
    Task<IEnumerable<T>> GetAllAsync();

    2 references
    Task<T> AddTAsync(T item);

    2 references
    Task<T?> GetByIdAsync(string id);

    2 references
    Task UpdateAsync(T item);

    2 references
    Task DeleteAsync(string id);
}
```

נסיון למימוש הממשק:
אחרי ההמרה מT Movie לT

```
public abstract class Repository<T>(IMongoService mongo) : IRepository<T>
{
    //private props:
    private readonly IMongoCollection<T> _collection = mongo.GetCollection<T>();
```

2 references

```
public async Task<T> AddAsync(T item)
{
    await _collection.InsertOneAsync(item);
    return item;
}
```

בעיה – לא לכל אובייקט יש Id
(אמנם לכל האובייקטים שלנו כן
יש Id – אך לא לכל האובייקטים
בעולם וT זה למעשה כל אובייקט)

2 references

```
public async Task DeleteAsync(string id)
{
    await _collection.DeleteOneAsync(m => m.Id == id);
}
```

פתרון – נגדיר שאנו מעוניינים לעבוד רק עם אובייקטים שיש להם Id

4 references

```
public interface IIdentity
{
    7 references
    string? Id { get; set; }
}
```

ניצור ממשק חדש
שמחלקה שמימשה אותו יש לה Id

פתרון – נגדיר שאנו מעוניינים לעבוד רק עם אובייקטים שיש להם Id

```
public abstract class Repository<T>(IMongoService mongo) : IRepository<T> where T: IIdentity
{
    //private props:
    private readonly IMongoCollection<T> _collection = mongo.GetCollection<T>();

    2 references
    public async Task<T> AddAsync(T item)
    {
        await _collection.InsertOneAsync(item);
        return item;
    }

    2 references
    public async Task DeleteAsync(string id)
    {
        await _collection.DeleteOneAsync(m => m.Id == id);
    }
}
```

המחלקה Repository מגבילה את האפשרויות של T רק לאובייקטים שיש להם ID

קוראים לזה Type Narrowing

ונעשה אותו דבר גם בממשק IRepository גם הוא מיועד לעבוד רק עם אובייקטים שיש להם ID

יש לשנות את רמת הנגישות של השדה Collection למקרה שנרצה לבצע שינויים בירושה ל'protected' יש להוסיף לכל המתודות virtual כדי שנוכל לדרוס אותן במידת הצורך.

יתר הקוד במחלקה IRepository<T>

```
public async Task<IEnumerable<T>> GetAllAsync()
{
    var cursor = _collection.Find(_ => true);

    //execute the find:
    var items = await cursor.ToListAsync();

    return items;
}
```

2 references

```
public async Task<T?> GetByIdAsync(string id)
{
    return await _collection.Find(m => m.Id == id).FirstOrDefaultAsync();
}
```

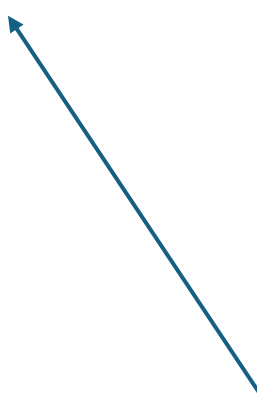
2 references

```
public async Task UpdateAsync(T item)
{
    await _collection.ReplaceOneAsync(m => m.Id == item.Id, item);
}
```


מחלקה אחרונה – סוף סוף קונקרטית ולא אבסטרקטית:

```
public class CardRepository(IMongoService service): Repository<Card>(service)
{
}
```

מחלקת עזר קטנטנה ונחמדה
היכן שנצטרך
IRepository<Card>
נוכל להשתמש בה 😊
(CardRepository cards)



נוסיף את הRepository הזה לקונטיינר של הDc

```
// Add services to the container.
```

```
builder.Services.AddSingleton<IMongoService, MongoService>();  
builder.Services.AddScoped<CardRepository>();
```

```
[ApiController]  
[Route("[controller]")]
```

0 references

```
public class CardsController(CardRepository cards) : ControllerBase  
{  
    [HttpGet]  
    0 references  
    public async Task<IActionResult> Get()  
    {  
        var all = await cards.GetAllAsync();  
        return Ok(all);  
    }  
}
```

```
[HttpGet("{id}")]
```

1 reference

```
public async Task<IActionResult> GetById(string id)
{
    var card = await cards.GetByIdAsync(id);
    return Ok(card);
}
```

שיעורי בית:
לחזור על הנושאים הבאים:

א) מתודות הרחבה – extension method

ב) Dtos למה צריך אותם?

ג) מה זה מחלקה abstract

ד) מתודה וירטואלית

ה) Generics - למה צריך? איך יוצרים מחלקה גנרית פשוטה כדי להחליף טיפוס לקוד קיים

ממליץ לראות את ההקלטה עם ה Repository ולממש אותו מספר פעמים כדי להבין את הניואנסים.

ממשו את המחלקה User

צרו לזה DTO לבקשות נכנסות

צרו לזה UserRepository

צרו קונטרולר עם 5 הפעולות שיש לנו ב repository עבור user

צרו קונטרולר עם 5 הפעולות שיש לנו ב repository עבור card