Module 1

Records - part I

Rationale

Building a basic web service

Let's imagine building a custom website for hosting **`Dotnetos Conference`**. The application should provide a **`REST endpoint`** that returns information about the conference speakers.

So, let's start with entity:

```
public class Speaker
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    public string Company { get; set; }

    [...]
}
```

Repository

To simplify, let's assume that **`in-memory collection`** is good enough to make our application up and running.

```
public class SpeakerRepository
{
    private readonly Speaker[] _speakersFrom2019 = {
        new Speaker {FirstName = "Adam", LastName = "Sitnik", Company = "Microsoft"},
        new Speaker {FirstName = "Matt", LastName = "Warren", Company = "Raygun", },
        new Speaker {FirstName = "Shay", LastName = "Rojansky", Company = "Microsoft"},
    };

    public Speaker[] GetAll() => _speakersFrom2019.ToArray();
}
```

Controller

What we need is a single controller with one **`GET endpoint`** that returns all speakers. By using injected **`SpeakerRepository`** we can ready all speakers.

```
[ApiController]
[Route("[controller]")]
public class SpeakersController : ControllerBase
{
    private readonly SpeakerRepository _repository;

    public SpeakersController(SpeakerRepository repository) => _repository = repository;

    [HttpGet]
    public IActionResult Get()
    {
        Speaker[] speakers = _repository.GetAll();
        return Ok(speakers);
    }
}
```

Controller

It is very bad.

Few problems:

- leaking full domain `Speaker` object to the end-client. It may expose internal properties, like personal address, email or id
- `Speaker` object may have references to the other properties, that can be lazy-loading entity living in another SQL table. It can cause sub-select queries during JSON serialization or failures
- any change to the `Speaker` entity may break the client-server contract

DTO - Data Transfer Object

It is an object that carries data between processes or applications, for example, a DTO object representing values provided by our web application.

A DTO object should not have other responsibilities other than carrying on the data and it should be **`immutable`** (readonly).

In ASP.NET application, a DTO object can be used to carry data from the backend to the frontend or a client application.

```
public class SpeakerDTO
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    public string Company { get; set; }
}
```

Controller

The speaker entity needs to be mapped (converted) to the DTO class. Many libraries like **`AutoMapper`** can do it automatically with reflection or source generators, but it can be also done manually. Sometimes it is the preferred way because it creates a strong reference between properties what makes this mapping more clear.

```
[HttpGet]
public IActionResult Get()
{
    Speaker[] speakerEntities = _repository.GetAll();
    SpeakerDTO[] speakers = speakerEntities.Select(x => new SpeakerDTO
    {
        FirstName = x.FirstName,
        LastName = x.LastName,
        Company = x.Company
    }).ToArray();

    return Ok(speakers);
}
```

Immutability

DTO Immutability

As mentioned before, a DTO class should be immutable. It prevents doing unwanted operations on the DTO level and reduces the chances of introducing a bug.

To make our DTO immutable, let's remove setters from the class by leaving our properties get-only. Initialization of the values will be done with a custom constructor that sets all values.

```
public class SpeakerDTO
{
    public SpeakerDTO(string firstName, string lastName, string company)
    {
        FirstName = firstName;
        LastName = lastName;
        Company = company;
    }

    public string FirstName { get; }

    public string LastName { get; }

    public string Company { get; }
}
```

DTO Immutability

Since now there are **`no setters`** and **`lack of default constructor`**, we need to adopt the mapping part to use new constructor

```
[HttpGet]
public IActionResult Get()
{
    Speaker[] speakerEntities = _repository.GetAll();
    SpeakerDTO[] speakers = speakerEntities
                            .Select(x => new SpeakerDTO(x.FirstName, x.LastName, x.Company))
                            .ToArray();

    return Ok(speakers);
}
```

Initialization of the DTO via constructor with parameters is supported by the **`Newtonsoft.JSON`** and **`System.Text.Json`** since .NET 5.

DTO Immutability

The approach of custom constructor and readonly properties provides Immutability, but it makes working with this entity complex. Sometimes we need a more flexible API that gives the user of that type more freedom, for example, to create an instance with fewer values if they are not needed at this moment.

```
public class SpeakerDTO
{
    public string FirstName { get; }
    public string LastName { get; }
    public string Company { get; }
    public bool IsMicrosoftMVP { get; }
    public string Bio { get; }
    public string GithubNickname { get; }
    public string TwitterNickname { get; }

    public SpeakerDTO(string firstName, string lastName, string company) { [...] }
    public SpeakerDTO(string firstName, string lastName, string company, bool isMicrosoftMvp, string bio, string git
    {
        [..]
    }
}
```

Easier usage

We can solve this problem in multiple ways:

- marking setters as `public`/`internal` and using `object initializer`
- creating `more constructors` when needed
- extending current constructor with `default parameters`

What if C# could help us with this boilerplate code by providing immutability together with user-friendly syntax?

Let's see C# 9 in action!