Udemy course: Ultimate ASP.NET pt. 5

Section 6: REST API Development Best Practices

- we want to cover several weaknesses or hiccups in our code
  - it is working, but is not good at some places
- examples:
  - it is discouraged to talk directly to the DB context from the controller
  - we want to introduce another layer of abstraction
  - another thing: we don't want to send over data objects with our API or receiving objects of that type
  - there is times when we don't want to send certain data back to the user
    - example: PUT request shouldn't return the ID usually
    - it can also harm the security of the application
- we want to discuss:
  - use of data transfer objects
  - and we will be refactoring our controller
    - so we don't have so much logic at this level of our application!
- so we will refactor this one controller and lay the foundation for the hotels controller

#26: Refactor POST Method with Data Transfer Object

- a Data Transfer Object is a model
- in our app we already have the Country Model
  - we don't want the data models to be what is beeing sent or received
- so we create an abstraction of that
- possible attack of an POST action, when to much data is beeing sent:
  - overposting attack, see link:
    - https://go.microsoft.com/fwlink/?linkid=2123754
    - link generated from Visual Studio controller scaffolding
  - to not have to pass the id, we will use a data transfer object
  - it would be called Viewmodel - in an MVC application with Userinterface
- in the POST example what field do I need:
  - just the Name and the ShortName!
  - and also probably not the whole list of Hotels
- in our example the Data Transfer object (DTO) could look like this:
  -
```
namespace HotelListingAPI.VSCode.Models.Country
{
    0 references
    public class CreateCountryDto
    {
        0 references
        public string Name { get; set; }
        0 references
        public string ShortName { get; set; }

    }
}
```
- Note in Swagger:
  - now it only shows name and shortname as request body sample
- when we would send the id in Postman, it would now be ignored; but request can still be successful
- in the data transfer object, you can also add your own validation
  - e.g. you can add [Required] to mark a field value is required
  - if the field is skipped or a empty string is in it, you will get error code 400 Bad Request
- we can now customize the experience that our user should be having without having to modify our data models
  - it follows the Single responsibility rule
    - the S in SOLID principle
- in this section we added the folder Models/Country with CreateCountryCto class

#27: Introducing AutoMapper

- we can install AutoMapper either via Nuget Package manager or dotnet install
- next we have to setup a configuration and inject it into our application
- we add Configuration/MapperConfig class
  - it inherits from Profile
  - uses AutoMapper

- we create a MapConfiguration
  - with:
    - ```
      public MapperConfig()

      {

      CreateMap<Country, CreateCountryDto>();

      }
      ```
  - actually we can write:
    - ```
      CreateMap<Country, CreateCountryDto>().ReverseMap();
      ```
    - since the order mustn't matter to us
- since we need it to be injectible, we have to register our configuration for our automapper
  - in Program.cs we add:
    - ```
      builder.Services.AddAutoMapper(typeof(MapperConfig));
      ```
  - with this we can inject AutoMapper anywhere
  - and then we can use it for mapping
- the aim of AutoMapper is to avoid writing code in this matter:
  - ```
    [HttpPost]
    0 references
    public async Task<ActionResult<Country>> PostCountry(CreateCountryDto createCountry)
    {
        var country = new Country{
            Name = createCountry.Name,
            ShortName = createCountry.ShortName,
        };
        _context.Countries.Add(country);
    ```
  - Note: the fields of the structure could be in the dozens
  - so we need to let the controller inject AutoMapper
  - and also let AutoMapper do the conversion for us
- the injection can be done quite easily
  - in the constructor of CountriesController
- ```
  1 reference
  private readonly IMapper mapper;

  0 references
  public CountriesController(HotelListingDbContext context, IMapper mapper)
  {
      this.mapper = mapper;
      _context = context;
  }
  ```
  - Note:
    we are using AutoMapper
    add IMapper in the contructor;
    Trick: we can select mapper and press Ctrl+. and then click "add field from parameter
  - when we have private fields: it is good practice to add an underscore at the beginning of the variable!
- so with this we can make our POST action more simple:
  - ```
    [HttpPost]
    0 references
    public async Task<ActionResult<Country>> PostCountry(CreateCountryDto createCountryDto)
    {
        // var country = new Country{
        //     Name = createCountry.Name,
        //     ShortName = createCountry.ShortName,
        // };
        var country = _mapper.Map<Country>(createCountryDto);
    ```
  - see we avoid the commented lines, and have one new line as replacement
- that is it about how we use AutoMapper!

#28: Refactor GET Methods

- in our GET action we will be using a Dto
- and the mapper
- make sure to map a list - not just a single country object!

- our new DTO defined as GetCountryDto
  - ```
    3 references
    public class GetCountryDto
    {
        0 references
        public int Id {get; set;}
        0 references
        public string Name { get; set; }
        0 references
        public string ShortName { get; set; }
    }
    ```
  - Note: we might need the id; for displaying purposes possibly
- our intermediate code:
  - ```
    // GET: api/Countries
    [HttpGet]
    0 references
    public async Task<ActionResult<IEnumerable<GetCountryDto>>> GetCountries()
    {
        // Select * from Countries,
        var countries = await _context.Countries.ToListAsync();
        // Note: we need a list, see return type!
        // AutoMapper do not alert about that!
        //var records = _mapper.Map<GetCountryDto(countries);
        var records = _mapper.Map<List<GetCountryDto>>(countries);
        return Ok(records);
    }
    ```
- also our mapper config needs to know, how to go from Country to GetCountryDto, so we add this line in MapperConfig.cs:
  - ```
    CreateMap<Country, GetCountryDto>().ReverseMap();
    ```
- then we will also refactor our get by id action
- what do we want here?
  - do we just want one country or
  - the country as well as the hotels?
- we go with just the country with all the details
  - two options we have
    - reuse the GetCountryDto created before
    - or do another Dto, we can declare in the same file as GetCountryDto
- so we are going with a new class GetCountryDtoDetails which also have a list of hotels
  - ```
    0 references
    public class GetCountryDtoDetails
    {
        0 references
        public int Id {get; set;}
        0 references
        public string Name { get; set; }
        0 references
        public string ShortName { get; set; }
        // public virtual IList<Hotel> Hotels { get; set; }
        0 references
        public List<GetHotelDto> Hotels{get; set;}
    }
    ```
  - Note: the hotels list here, map back to the Country model hotellist!
  - so when we run the query:
    - we get the list of hotels from the data
    - we do our mapping, it will automatically fill the list with the data coming from the database
  - rule of thumb
    - a Dto should never have a field which is directly related to a datamodel
    - only time when a dto crosses path with a data model is when we are doing a mapping operation!
    - as we have in the GET action above!
- next we rename GetCountryDtoDetails to just CountryDto
  - also we will create a HotelDto with the hotel fields
- Tipp:
  - to move classes to a own file, just select the class name and press Ctrl+. and select the appropiate action
- so back to refactoring the GET by ID action
  - Note: we also need to query the hotel details!
  - our new return type is CountryCto

- so we change this line
  - ```
    var country = await _context.Countries.FindAsync(id);
    ```
  - to
    ```
    var country = await _context.Countries.Include(q => q.Hotels)

    .FirstOrDefaultAsync(q => q.Id
    == id);
    ```
  - Include: queries for specific attributes, here specific Hotels
- so our revised GET by ID action looks like the following:
  - ```
    // GET: api/Countries/5
    [HttpGet("{id}")]
    0 references
    public async Task<ActionResult<CountryDto>> GetCountry(int id)
    {
        // var country = await _context.Countries.FindAsync(id);
        // now we need to also include the list of hotels!:
        var country = await _context.Countries.Include(q => q.Hotels)
            .FirstOrDefaultAsync(q => q.Id == id);

        if (country == null)
        {
            return NotFound();
        }
        var record = _mapper.Map<CountryDto>(country);
        return Ok(record);
    }
    ```

#29. Refactor Post method

- create a new Dto or can an existing Dto be resused?
  - we don't want to include Hotels -> so not GetCountryCto
  - the Id we need it -> so not CreateCountryCto
  - GetCountryDto has the id, but it should only be used for GET?!
    - I don't fully understand that!
- we could refactor GetCountryCto which comes closes to what we want
  - we create a BaseCountryCto
    - as an abstract class
    - since we inherit from it - several times
    - you cannot instantiate these
    - usually used for inheritance purposes
  - so GetCountryDto inherits from that new abstract class
  - the abstract Baseclass looks like this:
    - ```
      1 reference
      public abstract class BaseCountryDto
      {
          0 references
          public string Name { get; set; }
          0 references
          public string ShortName { get; set; }
      }
      ```
  - the base attributes can be removed in CreateCountryDto also, but the danger is that
    - that the validation rules would have to be kind of universal
    - if we redefine it here, it would override the base
    - but overall we just leave it, and inherit from BaseCountryDto
- finally we create an UpdateCountryDto for the PUT method, which inherits from the BaseCountryDto
  - very BAD would be to happen, that you don't enforce Dto to have a defined valid structure of fields
    - that is important for validation purposes
  - someone could update and mess up your data, if you don't enforce the structure
    - so that is why we use a BaseCountryDto
    - to replicate our validation
- we are putting a lot of efforts into all those files, why this hassle?
  - to enforce the S in the SOLID principle
  - which is separation of concerns
  - each file should have separation, one task, one meaning in life
- now we are changing the PUT action in the controller
  - we change the parameter (argument) to the UpdateCountryDto
      ```

    - then we have to find a country by that id in the Dto, or otherwise we can return NotFound()
        - ```
          var country = await _context.Countries.FindAsync(id);
          ```
    - if it is found: we can use a Mapper
        - it can do lots of magic for us
        - with this line:
          ```
          _mapper.Map(updateCountryDto, country);
          ```
        - it should take all the fields that map in update country Dto and update it in the country
        - the result which was just coming from the database
        - now we just have to let the database know, that it should save the data
        - this line is already there from before
    - Note: country in the above example is beeing tracked, it became an EntityState of Modified!
        - so the Mapper line automatically told entity framework, that we changed it, for the purpose to modify it
        - values from the left side are assigned to the right side
        - this is cool!
    - next we have to update our MapperConfig.cs and add a line there!
        - Note: likely error in case we forget that when we execute the PUT:
        - [ERR] An unhandled exception has occurred while executing the request.
        - AutoMapper.AutoMapperMappingException: Missing type map configuration or unsupported mapping.
        - with error response status 500
- Tipp:
    - Ctrl+C+K to comment quickly

#30: Implement Repository Pattern - Part 1

- purpose of a repository
    - create another layer of abstraction between our controller and the intelligence
    - intelligence means:
        - what is happening inside our controllers, inside our actions
    - the controller
        - we don't want it to be so involved in decision making
        - it is supposed to receive a request, route the request and receive data,
    - controller is not supposed to necessarily know how the data was gotten or whatever formatting was applied, etc., it shouldn't have to care
    - but as it stands, our controller cares a lot, examples:
        - it known how to connect to the database:
          ```
          private readonly HotelListingDbContext _context;
          ```
        - also about automapper configuration
          ```
          public CountriesController(HotelListingDbContext context, IMapper mapper)
          {
              this._mapper = mapper;
              _context = context;
          }
          ```
        - about the queries, of e.g. the countries
        - the conversion with mapper
        - and the returns
- so the controller mustn't know all of this
    - we will create a repository that is going to act like the machine or business intelligence
- we create two new folder one with name Repository, the other Contracts
    - often it would be named IRepository or contracts and repository
- the Contract
    - represents the abstraction of a class
- the repository represents the implementation of that class
- so in Contract we have interfaces
    - one is the interface
      ```
      IGenericRepository<T>
      ```
    - where T represents our data objects
    - in form of country and hotel
- so our GenericRepository is in charge of communicating with the database on our behalf
- we are having it Generic to avoid repetition
    - since in Hotels and countries some things would get repeated
      ```

    - DRY
        - Don't repeat yourself
- our main contract of IGenericRepository.cs now looks like this:
    -
```
interface IGenericRepository<T> where T : class

{

Task<T> GetAsync(int? id);

Task<List<T>> GetAllAsync();

Task<T> AddAsync(T entity);

Task DeleteAsync(int id);

Task UpdateAsync(T entity);

Task<bool> Exists(int id);

}
```
    - this is not all we need
    - we need also "Mini contracts" afterwards
    - in a new interface of ICountriesRepository
    - since we are dealing with countries which needs country specific contracts
    - we can inherit from the IGenericRepository interface
        -
```
public interface ICountriesRepository : IGenericRepository<Country>{

}
```
- about the implementation
    - we implement all interface method in the Repository
    - for AddAsync we already saw how it worked in our controller - but it will be a bit different
    - so we also need the context for database access
    - we have to do this form of action, since we have to access the database
    - again we do that in the constructor, just like we had in our Controller
        - ctor tab to create the constructor
    - our AddAsync method looks like this:
        -
```
public async Task<T> AddAsync(T entity)

{

await _context.AddAsync(entity);

await _context.SaveChangesAsync();

return entity;

}
```
    - the AddAsync method is a special EntityFramework method, it automatically inserts a entity object, to the entity (databasetable) it belongs to
    - actually we could have used this method already before in our POST action, but we have/had there:
        -
```
_context.Countries.Add(country);
```
    - we could have used the async method instead with await
        -
```
await _context.Countries.AddAsync(country);
```
    - in our AddAsync method:
        - if the entity object wouldn't belong, it would throw an error

        - after we Save the Changes and return the entity
- the others are:
    - ```
      public async Task DeleteAsync(int id)
      ```
    - ```
      public async Task<bool> Exists(int id)


      public async Task<List<T>> GetAllAsync()



      public async Task<T> GetAsync(int? id)



      public async Task UpdateAsync(T entity)
      ```
    - all these follow also this pattern
        - the easier ones:
        - GetAllSync (just see code in git coderepository)
          > we use Set<T> Method
          > go to the database and get the DB set accociated with T
        - GetAsync
          > if id is 0: return
          > we use FindAsync method and return the result
        - UpdateAsync
          > a bit different than what we saw before; fairly easy though
          > we use the Update method and SaveChangesAsync
        - Exists
          > we use GetAsync and Return the entityresult != null; (to return just a boolean)
        - DeleteAsync
          > we use Remove method
          > Note: not all method have async variants, like Remove
        - Update also cannot happen asynchronously
- a short recap
    - we are stating our Contract, do declare what we are capable of
    - then we implement our contract
    - we don't want our Controller to be very involved with our database, automapper, queries etc.
    - in next section we will modify all of this - controller

#31: Implement Repository Pattern - Part 2

- we created the GenericRepository Implementation
- now this section is about the CountriesRepository
- first the class definition - just according to the interface
    - ```
      public class CountriesRepository : GenericRepository<Country>, ICountriesRepository{}
      ```
    - Note: it will use any implementation of these two - either GenericRepository<Country> or ICountriesRepository
- our constructor in here and context:
    - ```
      private readonly HotelListingDbContext _context;



      public CountriesRepository(HotelListingDbContext context) : base(context)

      {

      this._context = context;

      }
      ```

- we also update our Program.cs slightly
  - of our builder.Service we need to use the AddScoped Method like this:

```
builder.Services.AddScoped(typeof(IGenericRepository<>), typeof(GenericRepository<>));

builder.Services.AddScoped<ICountriesRepository, CountriesRepository>();
```

  - to register the Repositorypattern
  - Note: first the interface, then the implementation in the code above
- now we have a specific repository
  - that we can extend beyond the basic CRUD operation
- in the CountriesRepository we will implement the GetDetails in a next section

#32: Refactor Controller to Use Repository

- now we have the repository implementations, we can actually do the refactoring of the CountryController
- we start by injecting a copy of our ICountriesRepository
  - we also removed the context
  - and insert a readonly field (Note: press Ctrl+. and insert field)
    -
```
public CountriesController(IMapper mapper, ICountriesRepository countriesRepository)
```
    - the constructor of our CountriesController
  - where we still have context, we use our _countriesRepository instead
    - the readonly class variable created above
  - and use our appropiate methods
  - like:
    -
```
var countries = await _context.Countries.ToListAsync();
```
  - becomes to:
    -
```
var countries = await _countriesRepository.GetAllAsync();
```
- challange now: to do the next refactoring all myself...
  - my mistake or issues:
    - we still need the mapper - I was not sure of that
    - in PutCountry, we should still keep the exception handling
    - see my comments in code
  - in DeleteCountry we still query ID with GetAsync first, to check whether country id exists
  - in CountryExists we have to change that to an async Task<bool> method!
  -