Udemy course: Ultimate ASP.NET pt. 8

Value Added Features

#52: Add Logging

- logging especially important when you have multiple persons and anything is not working correctly
  - and they would have to check your code
- example exception of a log file:
  - An unhandled excpetion has occured while executing the request.
  - System.InvalidOperationException: Unable to resolve service for type 'HotelListing.API.Contracts.IAuthManager' while attempting to activate ...AccountController
  - if someone got a 500 error while logging in, it is not really an indication that an dependency injection error happened!
- what if people want to keep track what is actually happening?
  - is somebody attempting something?
  - and maybe that something has gone wrong?
  - to do log just anywhere in e.g. our controller, we can inject the logger.
  -
- code:
  - ```
    public class AccountController : ControllerBase
    {
        5 references
        private readonly IAuthManager _authManager;
        1 reference
        private readonly ILogger<AccountController> _logger;

        0 references
        public AccountController(IAuthManager authManager, ILogger<AccountController> logger)
        {
            this._authManager = authManager;
            this._logger = logger;
        }
    ```
  - in the controller we just add ILogger<AccountController> and assign it to the class field
  - e.g. in Register we can do some logging about Registration attempt
- we insert a try/catch block in our Register action
  - we also don't want to throw, instead we better return a Problem
  - like this:
    - ```
      return Problem(@"Something Went Wrong in
      ThreadExceptionEventArgs " +

      nameof(Register) +

      "- User Registration attempt for
      {apiUserDto.Email}", statusCode: 500);
      ```
- but to do it everywhere it gets a bit annoying
  - you log information,
  - more important to catch the errors, and also log the error
  - warning could be used for example when in a query and id e.g. country id is not found
    - ```
      _logger.LogWarning($"Record not
      found in {nameof(GetCountry)} with {id}.");
      ```
- actually instead of doing the logging in the controller, it could make sense to use logging in our repository code instead!
  - see code in Repository/AuthManager.cs
  - for ideas how logging could be done
- methods in logging
  - LogInformation
  - LogWarning
  - LogError
- general Notes:
  - you don't want to collect data, which are too sensitive
  - if logging email adresses is too sensitive in your case, don't do that

#53: Global Error Handling
      ```

- we always want to catch the error that is causing some issue
  - also we looked into logging, etc.
- the more of that we implement
  - the harder it might be to maintain and read the code
  - e.g. if statements can complicate the code
- in this lesson: global exception handling
  - instead of catching all over the place, we just try/catch globally
- we want to also create our own exception, for our specific situation
  - with code in a new folder Exceptions
  - example:
    - we want to throw a "Not Found" exception
  - the code of our NotFoundException class, could be like:

```
namespace HotelListingAPI.VSCode.Exceptions
{
    0 references
    public class NotFoundException : ApplicationException
    {
        0 references
        public NotFoundException(string name, object key) : base($"{name} with id ({key}) was not found")
        {

        }
    }
}
```

- now to make it globally we have to look on our middleware in Program.cs
  - middleware: subapplication that is beeing run on the pipeline
  - like this are middlewares:

```
app.UseHttpsRedirection();

app.UseCors("AllowAll");

app.UseAuthentication();
app.UseAuthorization();

app.MapControllers();

app.Run();
```

- we create our own subapplication in this pipeline which will look for exceptions beeing thrown and it will automatically asset what kind of exception it is; and give an appropiate return value from the API
- in new folder Middleware, we create a new class ExceptionMiddleware
  - ctor with RequestDelegate:
    - ReguestDelegate: whenever a Request comes in, this object embodies that request
    - it is going to hijack this request while it is beeing processed
  - 1) first iteration of code:

```
0 references
public class ExceptionMiddleware
{
    2 references
    private readonly RequestDelegate _next;

    0 references
    public ExceptionMiddleware(RequestDelegate next)
    {
        this._next = next;
    }

    0 references
    public async Task InvokeAsync(HttpContext context)
    {
        try{
            await _next(context);
        }
        catch(Exception){
            throw;
        }
    }
```

  - here in await _next(context):
    the InvokeAsync gets called; it has the context, which has all the information about the request; the potential response
  - we are awaiting the result of the next operation relative to the request
  - then we are watching to see, if we catch any Exception occurs

    - no matter which type
    - e.g. DB Null Exceptions
        - we have lots of Database operations happening
        - potential points of failure
        - we don't want to try/catch at every single place
    - 2) second iteration
        - ```
          catch(Exception ex){

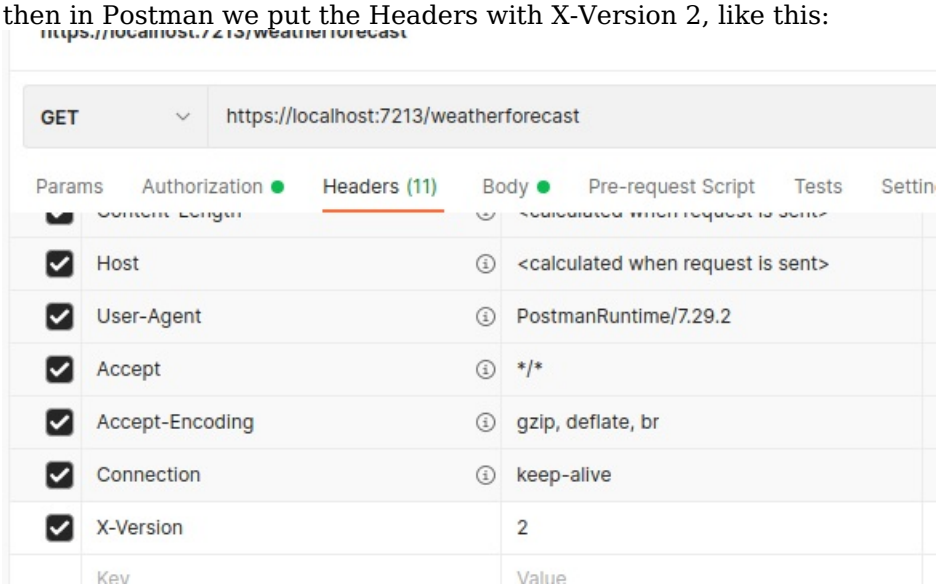
          await HandleExceptionAsync(context, ex);

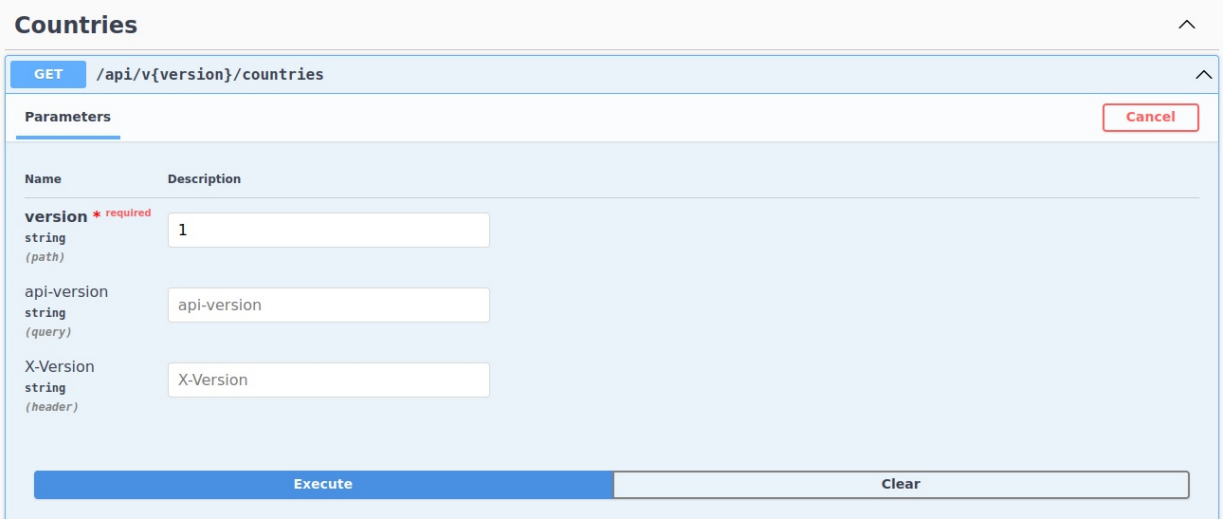
          }
          ```
    - now with the HandleExceptionAsync:
        - declaration about ContentType and StatusCode
        - as StatusCode we set the possibly worst error of InternalServerError
- we define a new ErrorDetails class
    - with string ErrorType and string ErrorMessage
    - ExceptionType and Message from Exception
- the HandleExceptionAsync method:
    - ```
      1 reference
      private Task HandleExceptionAsync(HttpContext context, Exception ex)
      {
          context.Response.ContentType = "application/json";
          HttpStatusCode statusCode = HttpStatusCode.InternalServerError;
          var errorDetails = new ErrorDeatils
          {
              ErrorType = "Failure",
              ErrorMessage = ex.Message,
          };

          switch (ex)
          {
              case NotFoundException notFoundException:
                  statusCode = HttpStatusCode.NotFound;
                  errorDetails.ErrorType = "Not Found";
                  break;
              case BadRequestException badRequestException:
                  statusCode = HttpStatusCode.BadRequest;
                  errorDetails.ErrorType = "Bad Request";
                  break;
              default:
                  break;
          }

          string response = JsonConvert.SerializeObject(errorDetails);
          context.Response.StatusCode = (int)statusCode;
          return context.Response.WriteAsync(response);
      }
      ```
- the more custom exceptions you might have, if you had one for Bad Request, one for whatever: it should have a own switch case
- so the ExceptionMiddleware is going to intercept the request going
    - it has a global track around every single request
- Note:
    - we return a nice JSON response in this method
    - that is also we have the ErrorDetails class for
    - this Middleware gives the possibility to standardize responses
- next we also add following line about the middelware in our Program.cs
    - method: app.UseMiddleware()
    - ```
      app.UseMiddleware<ExceptionMiddleware>();
      ```
        - with this it is including our Middleware in the whole pipeline!
- finally we can use our custom exception types to make use of it:
    - ```
      if (country == null)

      {
      ```

```
throw new NotFoundException(nameof(GetCountries), id);

}
```

- so now we don't have to worry about writing the same log message and if the log message changes we have to change it 20 times
- instead we just throw the exception and everything is happening kind of globally
  - if we want to change it, we just change it at one place in our ExceptionMiddleware
- Note also:
  - we can better make sure to getting appropriate error response for Bad Requests each time

#54: Implementing API Versioning

- important aspect of API development
  - especially APIs that are evolving over time
  - breaking changes should come with a new version
  - e.g. format of data changes, format of requests or data beeing returned
- in this case we should also document the changes to the customer (or follow developers)
- we install following packages for that documentation purposes:
  - Microsoft.AspNetCore.Mvc.Versioning
  - Microsoft.AspNetCore.Mvc.Versioning.ApiExplorer
  - $ dotnet add package Microsoft.AspNetCore.Mvc.Versioning
- we add the following lines to Program.cs to add the service to our Program.cs

```
builder.Services.AddApiVersioning(options =>
{
    options.AssumeDefaultVersionWhenUnspecified = true;
    options.DefaultApiVersion = new Microsoft.AspNetCore.Mvc.ApiVersion(1, 0);
    options.ReportApiVersions = true;
    options.ApiVersionReader = ApiVersionReader.Combine(
        new QueryStringApiVersionReader("api-version"),
        new HeaderApiVersionReader("X-Version"),
        new MediaTypeApiVersionReader("ver")
    );
});
```


```
builder.Services.AddApiVersioning(options =>
{
   options.AssumeDefaultVersionWhenUnspecified = true;
   options.DefaultApiVersion = new Microsoft.AspNetCore.Mvc.ApiVersion(1, 0);
   options.ReportApiVersions = true;
   options.ApiVersionReader = ApiVersionReader.Combine(
       new QueryStringApiVersionReader("api-version"),
       new HeaderApiVersionReader("X-Version"),
       new MediaTypeApiVersionReader("ver")
   );
});
```


- about the options:
  - AssumeDefaultVersionWhenUnspecified
    set a default version, when no version is specified
  - DefaultApiVersion: in above example the first number is the major version, and the second the subversion
- below this, we add following code, to add an ApiExplorer:

```
builder.Services.AddVersionedApiExplorer(
    options =>
    {
        options.GroupNameFormat = "'v'VVV";
        options.SubstituteApiVersionInUrl = true;
    });
```


```
builder.Services.AddVersionedApiExplorer(
   options =>
   {
      options.GroupNameFormat = "'v'VVV";
      options.SubstituteApiVersionInUrl = true;
   });
```

- to document an API Version in our controller actions we can just add following tag above the specific action
  - ```
    [ApiVersion("1.0")]
    ```
  - or if we put it in the header it should be:
    X-Version
- then we can do requests using the version with the querystring found in above lines (for us: api-version)
- so test it
  - we put API Version 2.0 above our Weatherforecast class
  - then in Postman we put the Headers with X-Version 2, like this:
  - 

- another versioning format can be accomplished with a change in the routes, e.g. we change the route of weatherforecast to:
  - ```
    [Route("[v{version:apiVersion}/[controller]")]
    ```
- it makes sense to use new classes for major version
  - you could copy the old controller and put V2 in the class name
  - we would have to use different routes also!
- in the tag of the versioning you can put a deprecated flag:
  - ```
    [ApiVersion("1.0", Deprecated = true)]
    ```
- of our old controller we also change the route to Countries controller:
  - from:
    - ```
      [Route("api/[controller]")]
      ```
  - to
    - ```
      [Route("api/v{version:apiVersion}/countries")]
      ```
  - if we don't change the route - if it stays the same as the other, we get an error
    - with complains about ambigious versioning or similar
- we can test in Swagger
  - the try interface looks now like this:
      ```



- it will change a different request URL depending on version 1 or 2
- with version 1 it will show:
  - https://localhost:7213/api/v1/countries

#55: Implementing Caching

- .net Core has Cache functionality built-in
- pro and cons for caching on a software level vs caching on a hardware level (or network level)
  - cons (software level)
    - will put more load on the server
    - the more it has to store, the more memory it will use; the more powerful your machine needs to be
  - hardware level pros
    - better when lots of people browsing a website
    - imagine everyone of them, keeps on badgering the database
    - that is when we would put in response caching to take pressure off
    - so quicker response
- changes to do:
  - add following lines in our Program.cs

```
builder.Services.AddResponseCaching(options =>
{
   options.MaximumBodySize = 1024;
   options.UseCaseSensitivePaths = true;
});
```

- we 1024 we are saying, cache should use up to 1024 KB of data
- if useCaseSensitivePaths is true:
  - URLs are case sensitive then
- below the Middleware part in Program.cs we have more to add:

```
app.UseResponseCaching();

app.Use(async (context, next) =>
{
   context.Response.GetTypedHeaders().CacheControl =
      new Microsoft.Net.Http.Headers.CacheControlHeaderValue()
      {
         Public = true,
         MaxAge = TimeSpan.FromSeconds(10)
      };
   context.Response.Headers[Microsoft.Net.Http.Headers.HeaderNames.Vary] =
      new string[] { "Accept-Encoding" };

   await next();
});
```

- Note:
  - we should add it after CORS!
  - in this example we put in Middleware code directly in Program.cs
  - both ways are use just for demonstration
  - when we add Cache control to our response
    - there are certain header values
    - that are going to come back, so the receiver of the data known that it was coming from the cache as opposed to coming from fresh data
    - in the example cache data are kept just 10 seconds
    - you can change that timespan based on your APIs objectives and based on your needs
    - here we accept the header names to vary
      the cache response may vary in terms of the type
    - and we can Accept-Encoding
- so that's it about caching
  - quick way to introduce caching to your API
  - this is more useful for data which is not changing much
    - then you can also expand the cache time
  - Note:
    - this was to show how Caching basically works
    - in the Real World, it is more complicated
    - e.g. with Pagination following after, there are issues with Caching
      > see: https://www.udemy.com/course/ultimate-aspnet-5-web-api-development-guide/learn/lecture/31214848#questions/17387414

#56: Implement Paging

- Paging is very important in API development
  - especially when there are lots of data to look through
- start of by creating a new custom type
  - in Models
  - QueryParameters class
  - just a class with PageSize, StartIndex,
  - and PagedResult as another class
  - with TotalCount, PageNumer, Record number, the Pagination Items
- now we modify our GenericRepository to let it know there is a new expectation
  - we can also name it GetAllAsync but with QueryParameters as input
  - In our Interface we add:
    - Task<PagedResult<TResult>> GetAllAsync<TResult>(QueryParameters queryParameters);
  - in this part we introduce another Generic called TResult
- the implementation in Repository/GenericRepository.cs

```
public async Task<PagedResult<TResult>> GetAllAsync<TResult>(QueryParameters queryParameters)
{
    var totalSize = await _context.Set<T>().CountAsync();
    var items = await _context.Set<T>()
        .Skip(queryParameters.StartIndex)
        .Take(queryParameters.PageSize)
        .ProjectTo<TResult>(_mapper.ConfigurationProvider)
        .ToListAsync();

    return new PagedResult<TResult>
    {
        Items = items,
        PageNumber = queryParameters.PageNumber,
        RecordNumber = queryParameters.PageSize,
        TotalCount = totalSize
    };
}
```

- we first get the totalSize or count of the database entity entries; on the Set we can just use CountAsync
- the items query is a bit tricky:
  - before we get to the list, we have to build up this query
  - see more on such query - the other course on EntityFramework
- the StartIndex is about where the client states, that I start from
- PageSize: the number of items on one page

- Note: we can do the mapping right here on the Query!
    - the advantage: no we have our Dto already,
    - the query gets actually optimized to go over to SQL
    - the mapping is removing fields - since we don't need all data
    - so it is making the query a bit more efficient also, since it does not query all data
    - for that we have to inject the Mapper to the constructor!
    - then we can use the .ProjetTo with the Mapper to convert the Generic TResult to what we need
        - this is still in the MappingConfiguration
- so by the time it hits .ToListAsync():
    - it knows everything it needs to know
        - knows which table
        - how many records
        - how many it should take
        - how many columns
        - and just then it executes the query!
- of course you could implement Pagination a bit differently; e.g. there might be no need for PageNumber
- Challange:
    - try to make the methods in GenericRepository more Generic
    - they are generic because they facilicate any table at any time, but
    - we can methods make return a Generic
- now in the Countries Controller we can have a query countries action like this:

```
// GET: api/Countries/?StartIndex=0&pagesize=25&PageNumber=1 // -> how request has to look like
[HttpGet]
public async Task<ActionResult<PagedResult<GetCountryDto>>> GetPagedCountries([FromQuery] QueryParameters queryParameters)
{
    var pagedCountriesResult = await _countriesRepository.GetAllAsync<GetCountryDto>(queryParameters);
    return Ok(pagedCountriesResult);
}
```

    - so with the request: the query parameters have to be given
        - in the class QueryParameters it is
- and now there are some more refactoring possiblities:
    - for that we also inject the mapper in CountriesRepository
- and the Method for GetDetails we can modify from:

```
// this is old:
    public async Task<Country> GetDetails(int id)
    {
        return await _context.Countries.Include(q => q.Hotels)
            .FirstOrDefaultAsync(q => q.Id == id);
    }
// to new:
    public async Task<CountryDto> GetDetails(int id)
    {
        var country = await _context.Countries.Include(q => q.Hotels)
            .ProjectTo<CountryDto>(_mapper.ConfigurationProvider)
            .FirstOrDefaultAsync(q => q.Id == id);

        if (country == null)
        {
            throw new NotFoundException(nameof(GetDetails), id);
        }
        return country;
    }
```

- new refactoring in Countries Controller:
    - we change following:

```
[HttpGet("{id}")]
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 _countriesRepository.GetAsync(id);

        if (country == null)
        {
            //_logger.LogWarning($"Record not found in {nameof(GetCountry)} with {id}.");
            //return NotFound();
            throw new NotFoundException(nameof(GetCountries), id);
        }
        var record = _mapper.Map<CountryDto>(country);
        return Ok(record);
    }
```

to:

```
    // GET: api/Countries/5
    [HttpGet("{id}")]
    public async Task<ActionResult<CountryDto>> GetCountry(int id)
    {
        var country = await _countriesRepository.GetDetails(id);
        return Ok(country);
    }
```

- Note: we also might have to add GetDetails in our Interface still
  - and need the injections in HotelsRepository
- we get now a error of AmbigiousMatchException:
  - this comes from equal routes
  - we forgot to modify the route of the GetPagedCountries!
  - we just modify the GetAll Action route
  -
    ```
    [HttpGet("GetAll")]

    public async Task<ActionResult<IEnumerable<CountryDto>>> GetCountries()
    ```
- Note:
  - caching from previous lesson might influence the result -> it does not give the right results always
- two possible challanges:
  - 1) implement the Generics in the GenericRepository
  - 2) implement the pagination in the Hotels controller

#57: Exploring OData

- third party library
- can affix itself to an API and
  - accomplish ordering, sorting, searching
  - with minimal effort
  - it happens at query string level
  -
  -
- install package Microsoft.AspNetCore.OData
- we add the following in our Program.cs

```
builder.Services.AddControllers().AddOData(options =>
{
   options.Select().Filter().OrderBy();
});
```

  - for example we can add it, just after the caching and after adding scoped services
  - with the options we specify which options we want to enable; here we just use
    - Select, Filter and OrderBy
- we experiment with CountriesV2Controller this time
  - to start place this annotation above your GetCountries() query:
  -
    ```
    [EnableQuery]
    ```
- in Testing with Swagger:
  - we can append following to the query string, to only query the names:
    - ?$select=name
    - to add shortname just put name,shortname
  - in my case the query URL is like this:
    - https://localhost:7213/api/v2/countries?$select=name

    - we can also filter for specific result using following query
        - ?$filter=name eq 'Cuba'
        - would list all entries with Cuba as name
- if we want to order results:
    - $orderby
- so that is how you easily introduce filtering, ordering, sorting
- you can combine queries

#58: Project Architectural Changes

- we were able to built a fully functional API using the default architecture
- but as we increase the functionality as we increase number of controllers and assets that need to go into the API
    - it might get a bit cluster
- we might be better of splitting out this project, into a number of miniprojects or projectfolders for particular operations
- let's start with data related files
    - we move it to a own Subproject of type ClassLibrary
    - HotelListing.Data
        - the Model stuff, including migration files
- since I am making the steps in VS code, here is my steps:
    - $ dotnet new classlib -o HotelListing.Data
    - $ dotnet sln add HotelListing.Data
    - then for the version control we can move the desired files with git move
        - $ git mv ./Data/* HotelListing.Data
    - next:
        - we should add the reference to this project in the old project
        - there is a Solution tab in VS code where it shows references
        - below references -> projects -> Add project references it should automatically insert the subproject
        - actually it mustn't be installed anymore, it seems to work with adding the missing package references in the csproj file
- another subproject we will have is
    - Core
    - 3rd party library stuff, stuff which is shared between projects, etc
- for me in Visual Studio Code:
    - took much longer to get it running and every error taken out
    - kinda troublesome
- so to learn from this lesson:
    - have a good understanding of your architecture - if possible already at the start
    - in case you have to refactor: know how to

#59: Repository Refactor

- implement parts of the challange above
- more generic methods, to reduce some of the mapping
- many ways to accomplish things
    - sometimes there are good alternatives
    - or worse ones
    - some might be too complicated
    - let the context determine what to do
- the new refactored - competely new - actions suggested here:
    - Task<TResult> GetAsync<TResult>(int? id);
        - TResult represents the data type of the call GET,
        - the own TResult is also taken in
    - Task<List<TResult>> GetAllAsync<TResult>();
        - similar to GetAsync above
    - Task<TResult> AddAsync<TSource, TResult>(TSource source);
        - this is a bit more tricky!
        - instead of sending T entity remember, we did:

```
Task<T> AddAsync(T entity);
```

        - the central entity, instead
        - give me the T Source, because I want to have the DtoSource come into the respository
        - then from the Dto we know which Type should be returned, so we also put TResult
    - Task UpdateAsync<TSource>(int id, TSource source) where TSource : IBaseDto;

        -
- see the reference code on the repository.
- sometimes the complexity vs. the perceived efficiency may leave our code not very readable
    - you want to strike the balance between both
    - important to avoid multiple equal code several times; so use Generics, when it makes sense
    - try to have all the Mappings just in one place
    - as example how it can be done, see the GenericsRepository, UpdateAsync Action:

```
public async Task UpdateAsync<TSource>(int id, TSource source) where TSource : IBaseDto
{
    if (id != source.Id)
    {
        throw new BadRequestException("Invalid Id used in request");
    }

    var entity = await GetAsync(id);

    if(entity == null)
    {
        throw new NotFoundException(typeof(T).Name, id);
    }

    _mapper.Map(source, entity);
    _context.Update(entity);
    await _context.SaveChangesAsync();
}
// with following baseDto:
namespace HotelListing.VSCode.Models
{
    public interface IBaseDto
    {
        int Id { get; set; }
    }
}
```

    - DRY: Don't repeat yourself!
    - steps:
        - whatever the source, map it over
        - update the entity
        - then do the update, and save changes
- next about some refactoring in CountriesRepository, GetDetails Method

```
public async Task<CountryDto> GetDetails(int id)
{
    var country = await _context.Countries.Include(q => q.Hotels)
        .ProjectTo<CountryDto>(_mapper.ConfigurationProvider)
        .FirstOrDefaultAsync(q => q.Id == id);

    if (country == null)
    {
        throw new NotFoundException(nameof(GetDetails), id);
    }

    return country;
}
```

    - here new the ProjectTo with Mapping
        - we had to inject the Mapper in the constructor
        - if it is null, we throw our custom exception; else we return the result

#60: Controller Refactor

- refactoring done in countries controller - V1, and then the Hotels controller
    - we are retaining the version numbers, since
- the controller does not need to know how it is getting, what it is getting
    - it only needs to know, I am retrieving data and returning the data

- all the mapping and logic, we want to reduce in that controller
- our get country details, we can reduce from that:

from:

```
// GET: api/Countries/5
[HttpGet("{id}")]
public async Task<ActionResult<CountryDto>> GetCountry(int id)
{
    var country = await _countriesRepository.GetAsync(id);

    if (country == null)
    {
        throw new NotFoundException(nameof(GetCountries), id);
    }
    var record = _mapper.Map<CountryDto>(country);
    return Ok(record);
}
```

to:

```
// GET: api/Countries/5
[HttpGet("{id}")]
public async Task<ActionResult<CountryDto>> GetCountry(int id)
{
    var country = await _countriesRepository.GetDetails(id);
    return Ok(country);
}
```

  - Note:
    - the new refactored code is much shorter, less lines
    - much easier, no more mapping, no more if statement required
  - also other methods were refactored; like DeleteCountry also now shorter and easier; PostCountry we also delete one line of mapping
  - also see how the PutCountry method is now shorter after refactoring
    - no more mapping and only on one place a CountriesRepository method is used; following: `await _countriesRepository.UpdateAsync(id, updateCountryDto);`
- same style refactoring were prepared in the HotelController
  - no more mappings and use of the new methods, which are already implementing the mapping
- Note:
  - we implemented a new BaseDto before
  - our HotelDto and UpdateCountryDto now also inherits from IBaseDto
  - IBaseDto just has a Id property
- refactoring helps you
  - maintain your standards
  - not repeat yourself
  - and make your code more readable
  - if you have to compromise performance and readability, then you should consider which is more important

#61: Add JWT Authetication to Swagger Doc

- we want to customize our Swagger, so that there is a better documentation of the Actions
  - we will be adding titles, and human readable information
- we also want to be able to test the JWT Authentication
  - even if it is not proteced by the bearer token itself
  - so we don't have to use Postman
- in our Program.cs there is the AddSwaggerGen() Method we have to expand that with options

```
builder.Services.AddSwaggerGen( options => {
    options.SwaggerDoc("v1", new Microsoft.OpenApi.Models.OpenApiInfo { Title = "Hotel Listing API",
Version = "v1"});
    options.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme{
        Description = @"JWT Authorization header using the Bearer scheme.
                        Enter 'Bearer' [space] and then your token in the text input below.
                        Example: 'Bearer 12345abcdef'",
        Name = "Authorization",
        In = ParameterLocation.Header,
```

```
        Type = SecuritySchemeType.ApiKey,
        Scheme = "Bearer"
    });
}
);
```

    - when we now restart our server, we see v1 next to the API Title (Hotel Listing API)
    - also there is a new Authorize button
    - Note: so far however it is not possible to authorize on Swagger yet, so far it is only documented on Swagger
- to actually add the Bearer Login Functionality we add the following:

```
options.AddSecurityRequirement(new OpenApiSecurityRequirement
{
    {
        new OpenApiSecurityScheme
        {
            Reference = new OpenApiReference {
                Type = ReferenceType.SecurityScheme,
                Id = "Bearer"
            },
            Scheme = "0auth2",
            Name = "Bearer",
            In = ParameterLocation.Header
        },
        new List<string>()
    }
});
```

- now after restarting our server and refresh Swagger page, we see locks on the right-hand side of each endpoint
- Note:
    - about the repository code: somewhere in my recent commits I broke the route of AccountController, so I didn't have the Login and Register actions in Swagger
    - so the route was corrected back to `[Route("api/[controller]")]`
        - also I had to add `[ApiController]`
- so like this we are now able to login in Swagger, via the token result from the login action
- Note:
    - we can replace the hardcoded string "Bearer" with the magic string JwtBearerDerfaults.AuthenitcationScheme
    - to reduce risk of spelling errors