Application Architecture

IT@AP

Sven Mariën

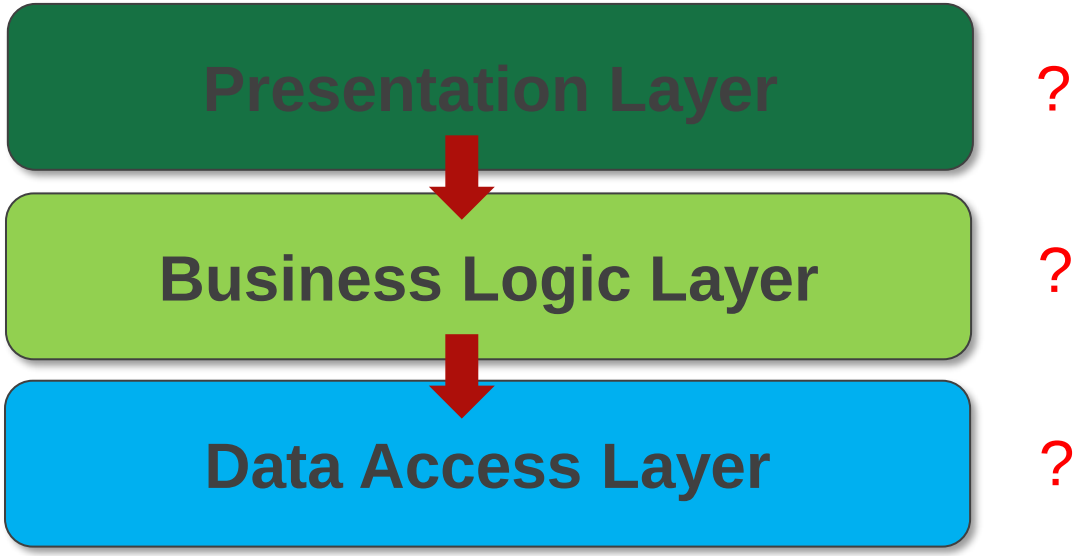
(sven.marien01@ap.be)

AP HOGESCHOOL ANTWERPEN

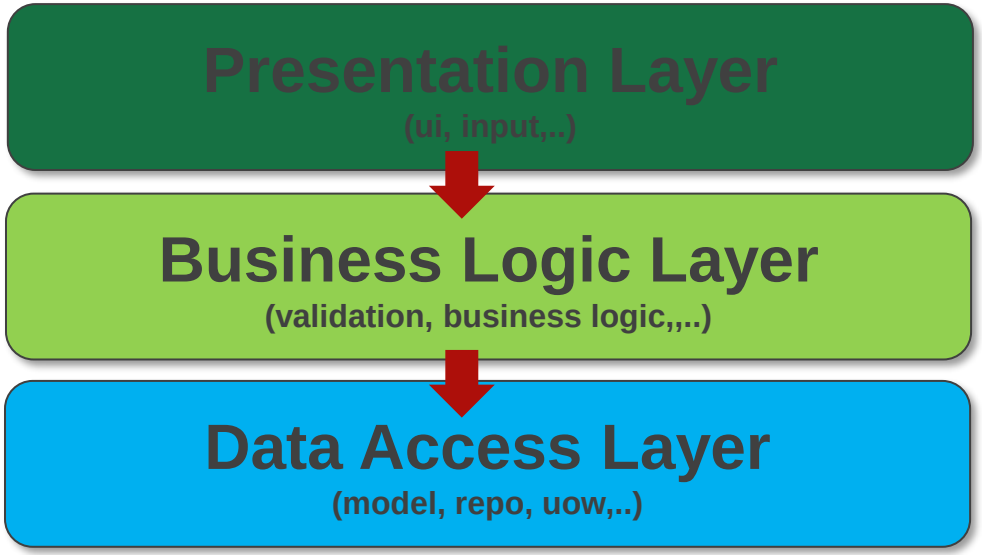
AP.BE

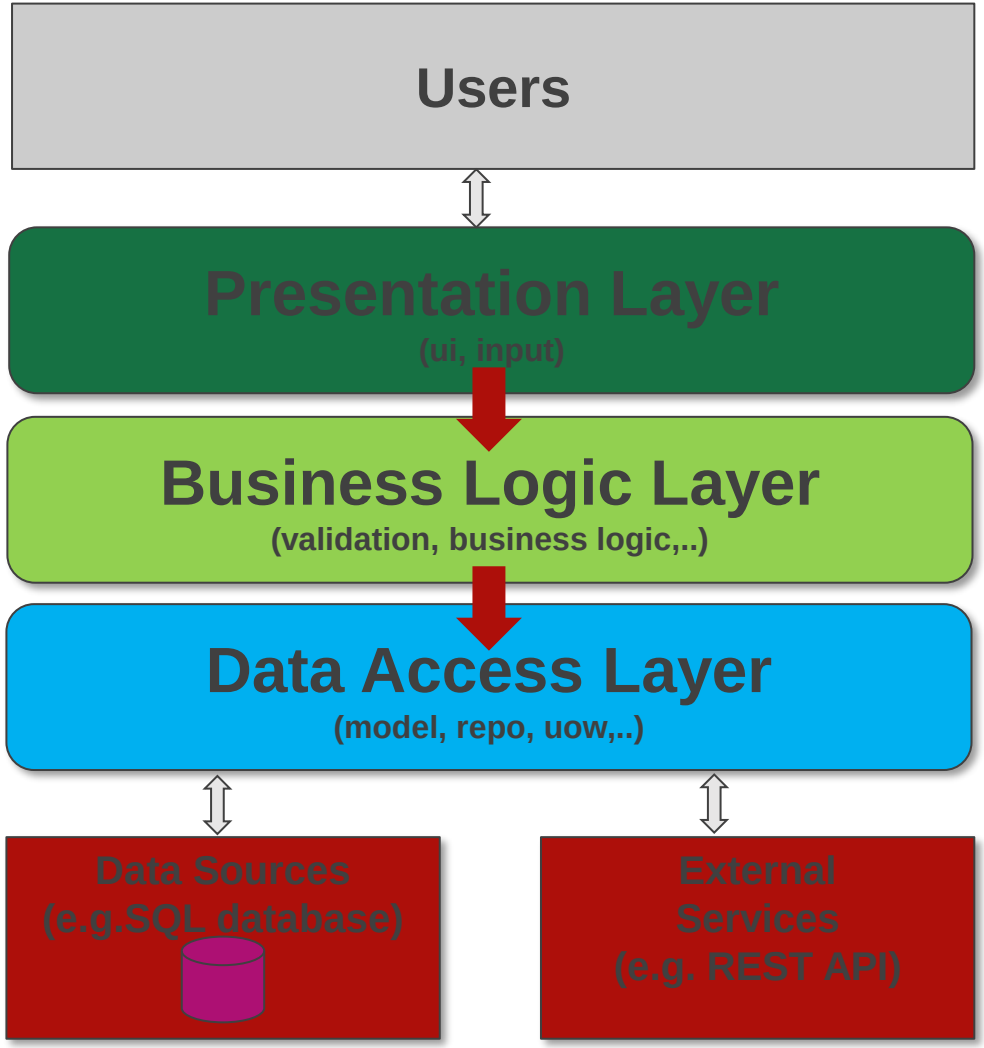
3-layer model

- = basic architecture



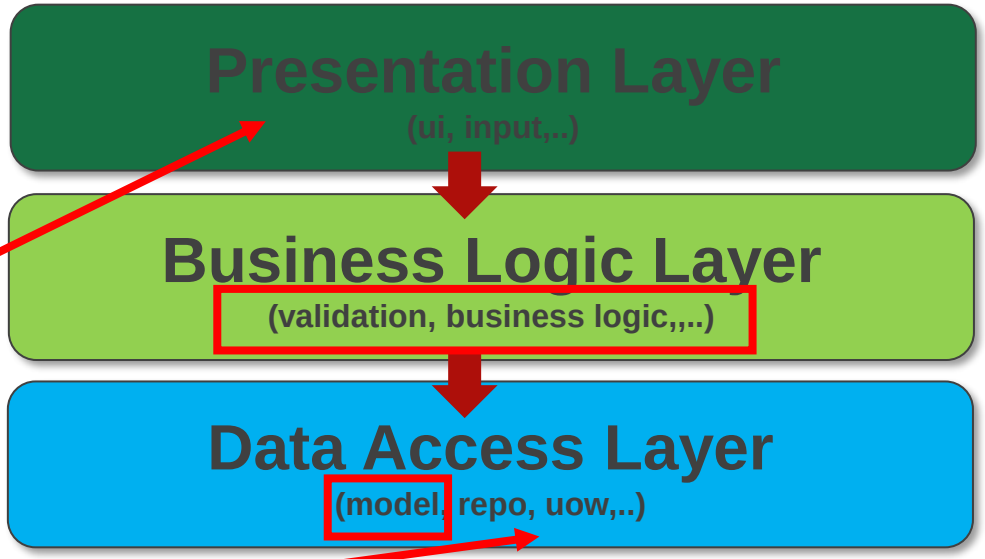
3 layer model




Users
Presentation Layer
(ui, input)
Business Logic Layer
(validation, business logic,..)
Data Access Layer
(model, repo, uow,..)
Data Sources
(e.g.SQL database)
External
Services
(e.g. REST API)

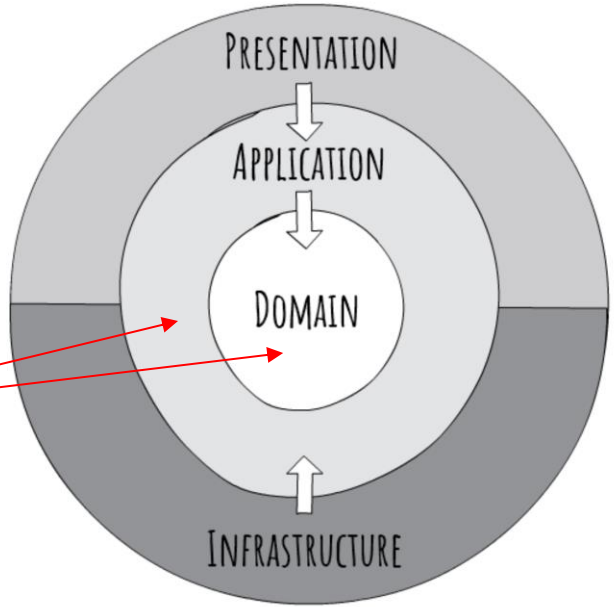
Clean Architecture

- The real "**core**" of the application is
  - **Business logic**
  - **Validation logic**
  - **Model (entities)**
- How/where the data is presented is **not** part of the core.
- How/where the data is stored is **not** part of the core.



Clean architecture

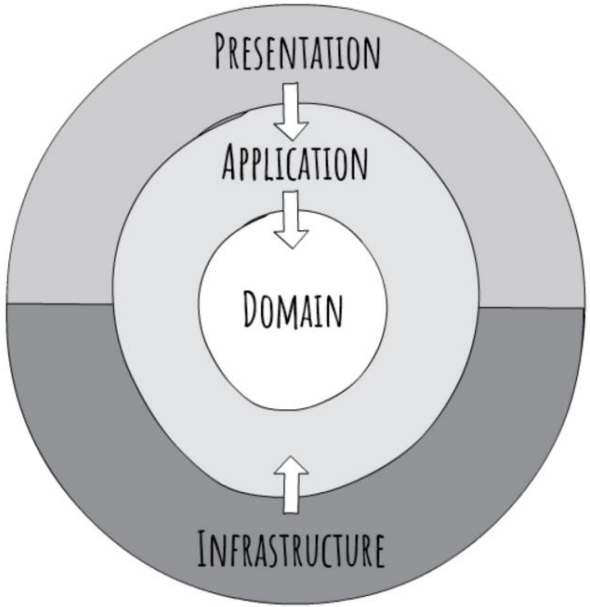
- Other flavors:
  - Onion architecture
  - Hexagonal architecture
  - ..



Application core

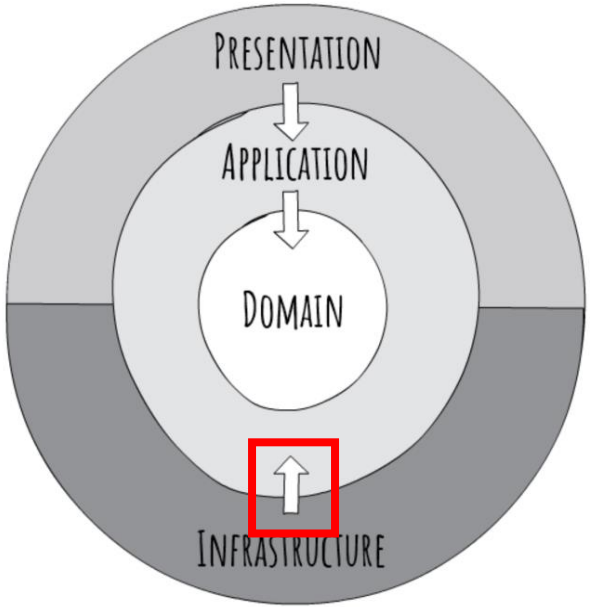
Clean architecture

- Application Core (Application+ Domain)
  - Business logic
  - Validation logic
  - Domain Model (Entities)
- Infrastructure
  - Gateway to the outside world
  - Repository, Unit of Work,..
  - External API's



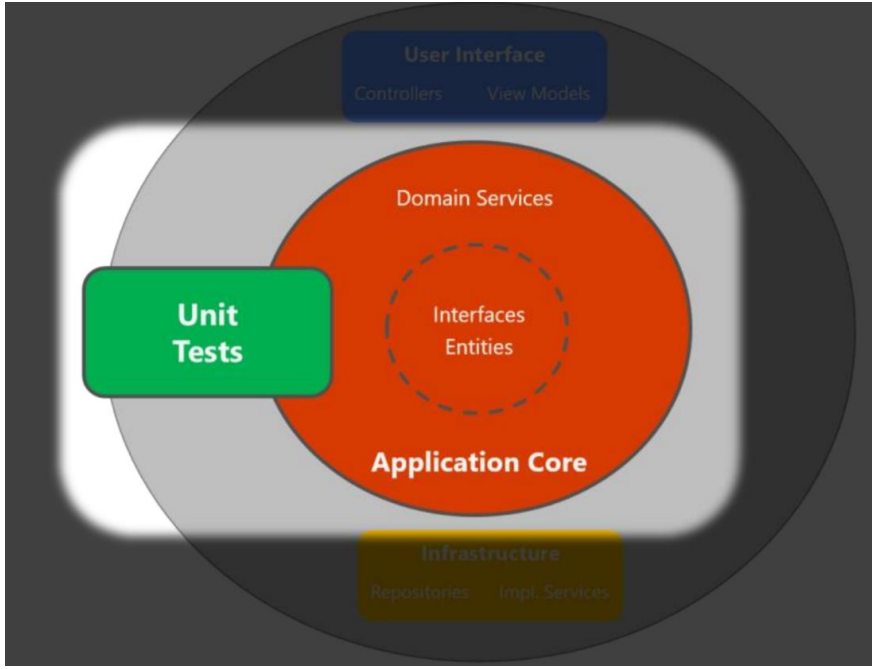
Clean architecture

- Check the depencies….
  - Application layer wants to retrieve store an entity..
  - Application layer wants to send an email
  - Application layer wants to call an external API
  - …
- = Dependency Inversion principle !
  - Application layer defines the interfaces
  - Infrastructure layers contains the implementations



Clean architecture

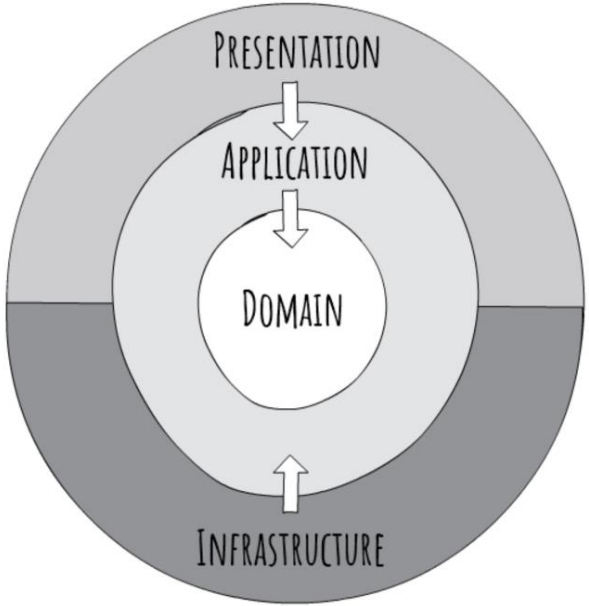
- Core
  - Has no dependencies !
  - Easily testable

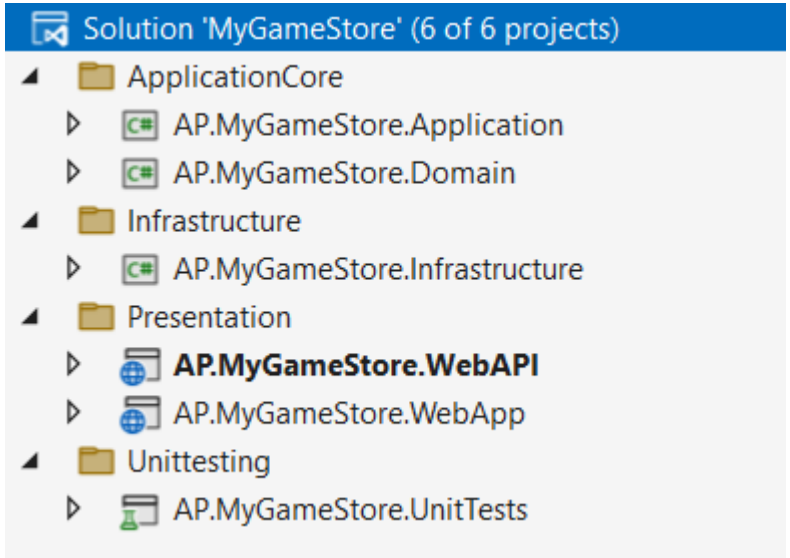


Step by step towards Clean Architecture

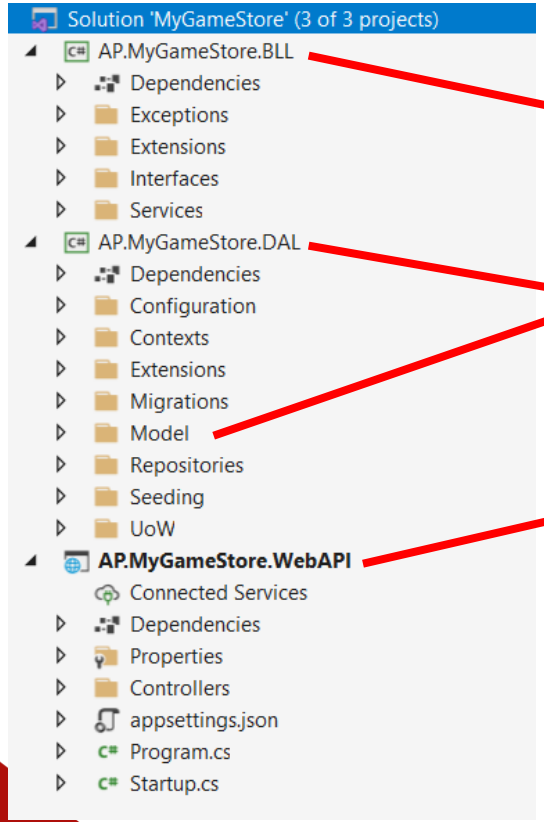
1. Setup projects & dependencies
2. Add EF migrations
3. Repository pattern
4. Async/await
5. CQRS + Mediator pattern
6. DTO/Viewmodels + Mapper
7. Status codes
8. Validation

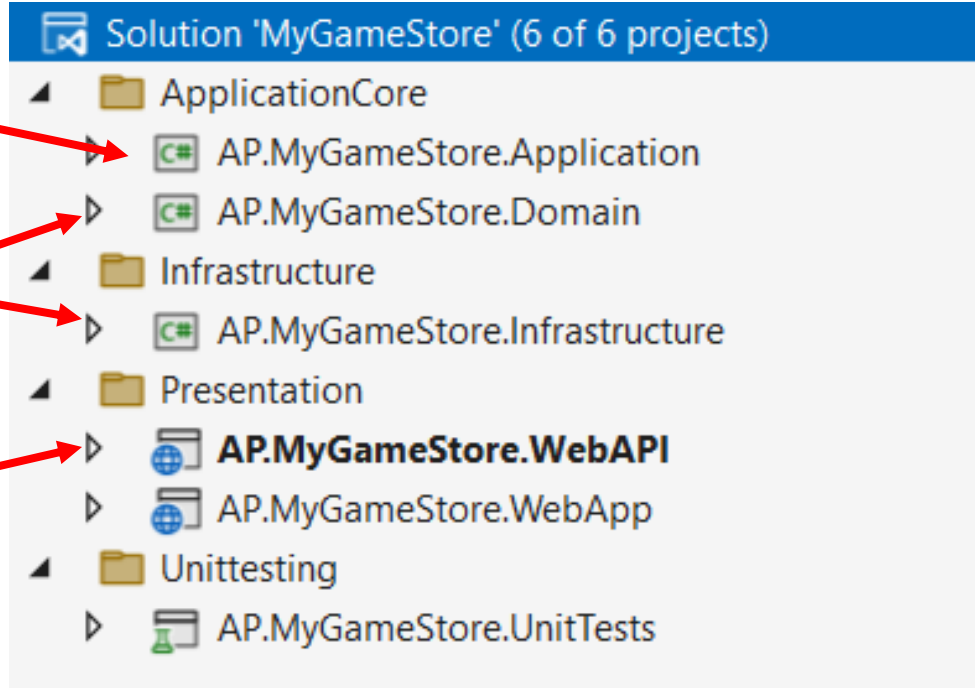
Step 1: Setup projects & dependencies



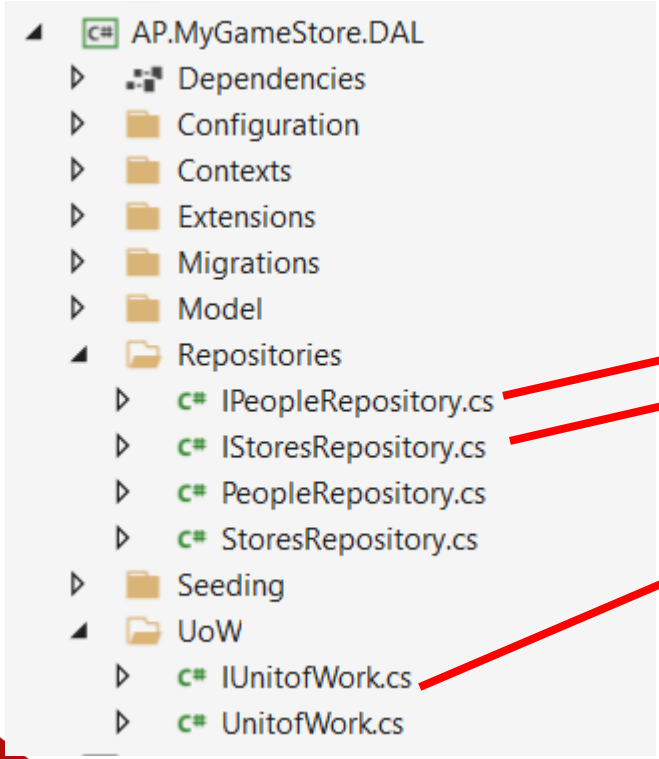


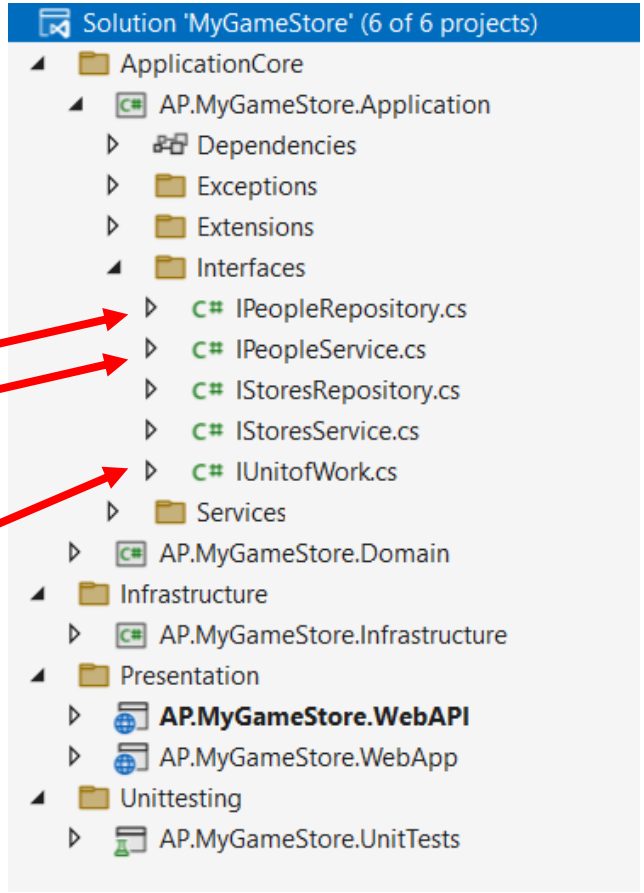
Step 1: Setup projects & dependencies





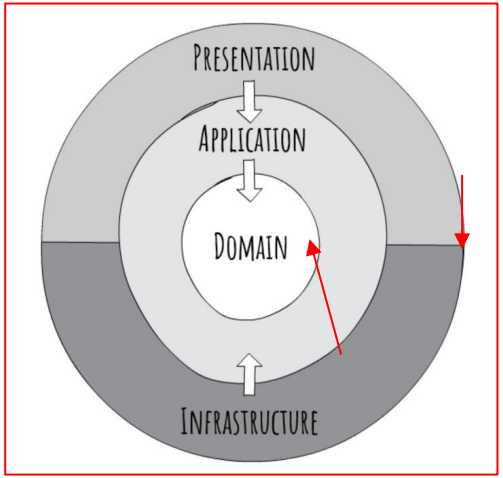
Step 1: Setup projects & dependencies

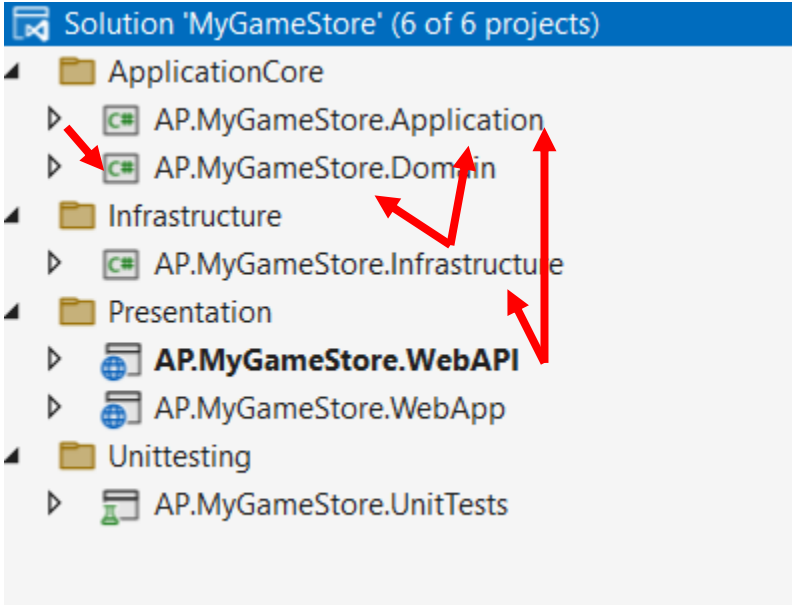




Step 1: Setup projects & dependencies

- Set Project dependencies





Step 1: Setup projects & dependencies

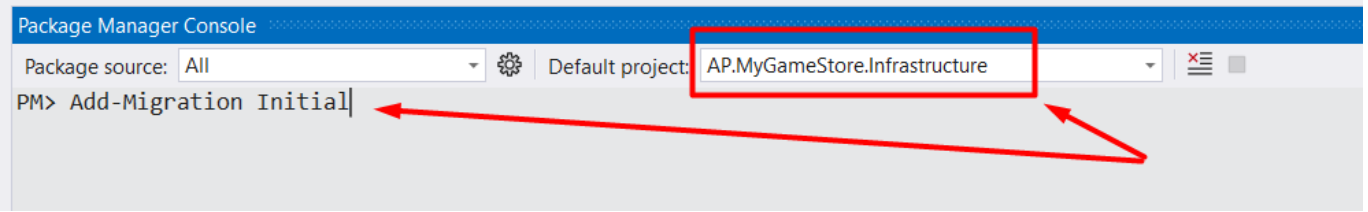
- Startup

```
0 references
public void ConfigureServices(IServiceCollection services)
{
    services.RegisterDAL();
    services.RegisterServices();
    services.AddControllers();
    services.AddSwaggerGen(c =>
    {
```

```
0 references
public void ConfigureServices(IServiceCollection services)
{
    services.RegisterInfrastructure();
    services.RegisterApplication();
    services.AddControllers();
    services.AddSwaggerGen(c =>
    {
```

Step 2: Migrations

- Migrations can be created and applied from the Package Manager Console



Step 3: Repository Pattern

- Room for optimization ?

```
6 references
public interface IPeopleRepository
{
    2 references
    IEnumerable<Person> GetAll(int pageNr, int pageSize)

    4 references
    Person GetById(int id);
    2 references
    Person Create(Person newPerson);

    1 reference
    Person Update(Person modifiedPerson);
    2 references
    void Delete(Person person);

    2 references
    bool AnyWithEmployer(int employerId);

    3 references
    IEnumerable<Person> GetByEmployer(int employerId);

    2 references
    void Delete(IEnumerable<Person> people);
```

```
6 references
public interface IStoresRepository
{
    2 references
    IEnumerable<Store> GetAll(int pageNr, int pa

    6 references
    Store GetById(int id);
    3 references
    Store GetByName(string name);
    2 references
    Store Create(Store newStore);

    1 reference
    Store Update(Store modifiedStore);
    2 references
    void Delete(Store store);

}
```

Step 3: Repository Pattern

- Generics

```
public interface IPeopleRepository
{
    2 references
    IEnumerable<Person> GetAll(int pageNr, int pageSize);

    4 references
    Person GetById(int id);
    2 references
    Person Create(Person newPerson);

    1 reference
    Person Update(Person modifiedPerson);
    2 references
    void Delete(Person person);

    2 references
    bool AnyWithEmployer(int employerId);
```

```
8   {
    1 reference
9   public interface IGenericRepository<T>
10  {
    0 references
11      IEnumerable<T> GetAll(int pageNr, int pageSize)
12
    5 references
13      T GetById(int id);
    1 reference
14      T Create(T newPerson);
15
    0 references
16      T Update(T modifiedPerson);
    1 reference
17      void Delete(T person);
    }
```

```
6 references
public interface IPeopleRepository : IGenericRepository<Person>
{
    2 references
    bool AnyWithEmployer(int employerId);

    3 references
    IEnumerable<Person> GetByEmployer(int employerId);

    2 references
    void Delete(IEnumerable<Person> people);

}
```

Step 4: async/await

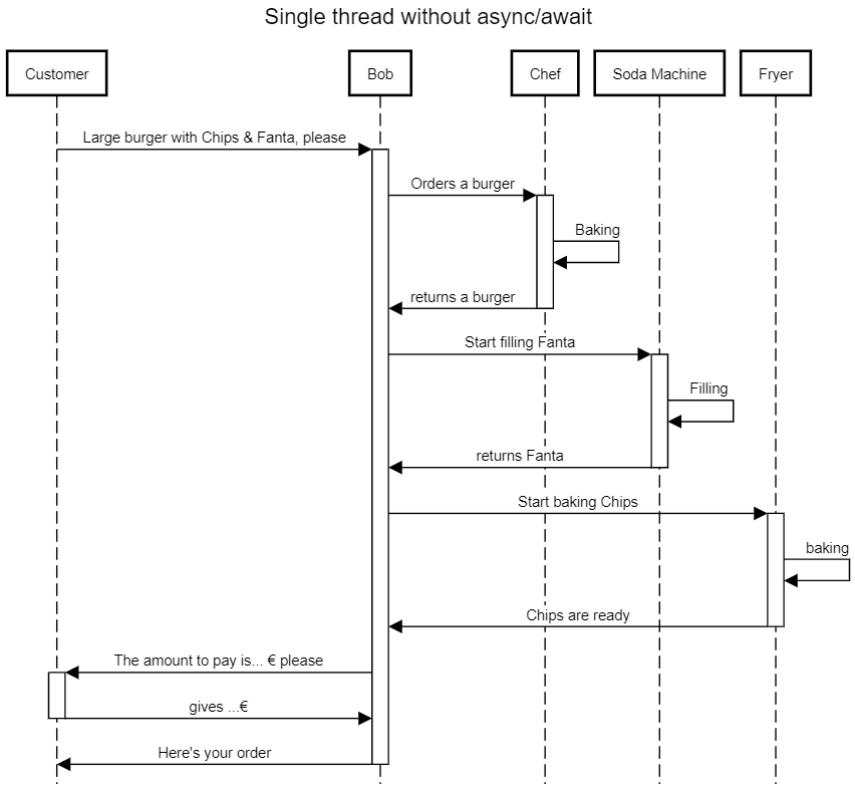
- Without async/await:
  - Every call will block a thread until it is finished
  - Eg. long queries to db, long network requests to other API's....
- With async/await
  - A Thread has time to start handling other incoming requests while previous requests are in progress
- https://www.carlrippon.com/scalable-and-performant-asp-net-core-web-apis-asynchronous-operations/

Step 4: async / await

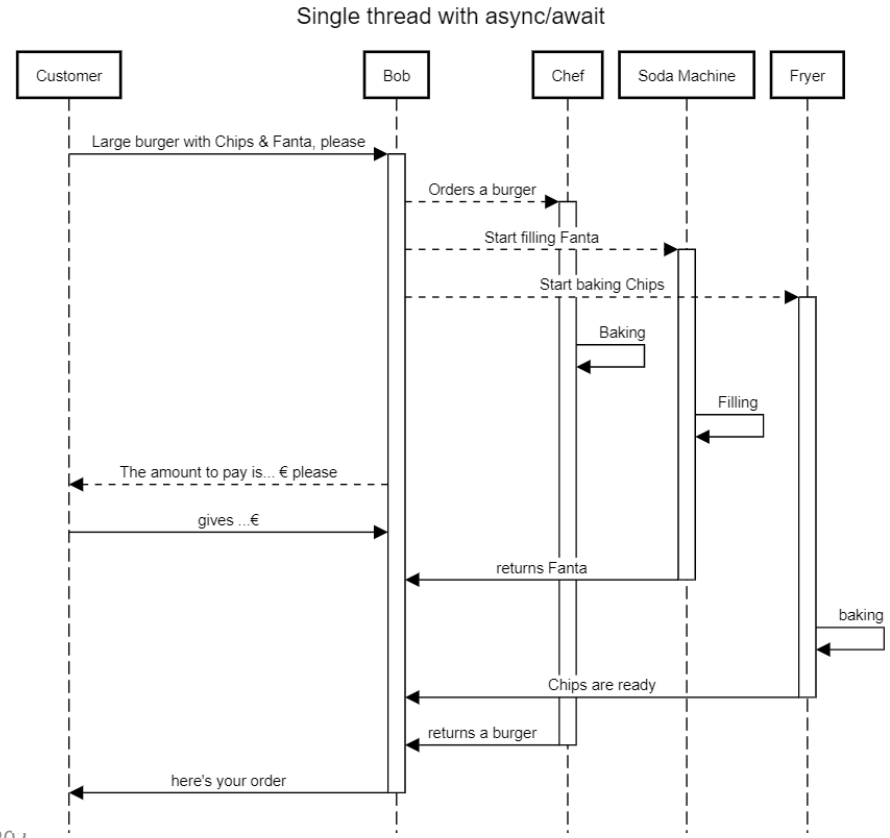
- Single-threading versus multi-threading:

But: Thread pool is limited in number of threads !

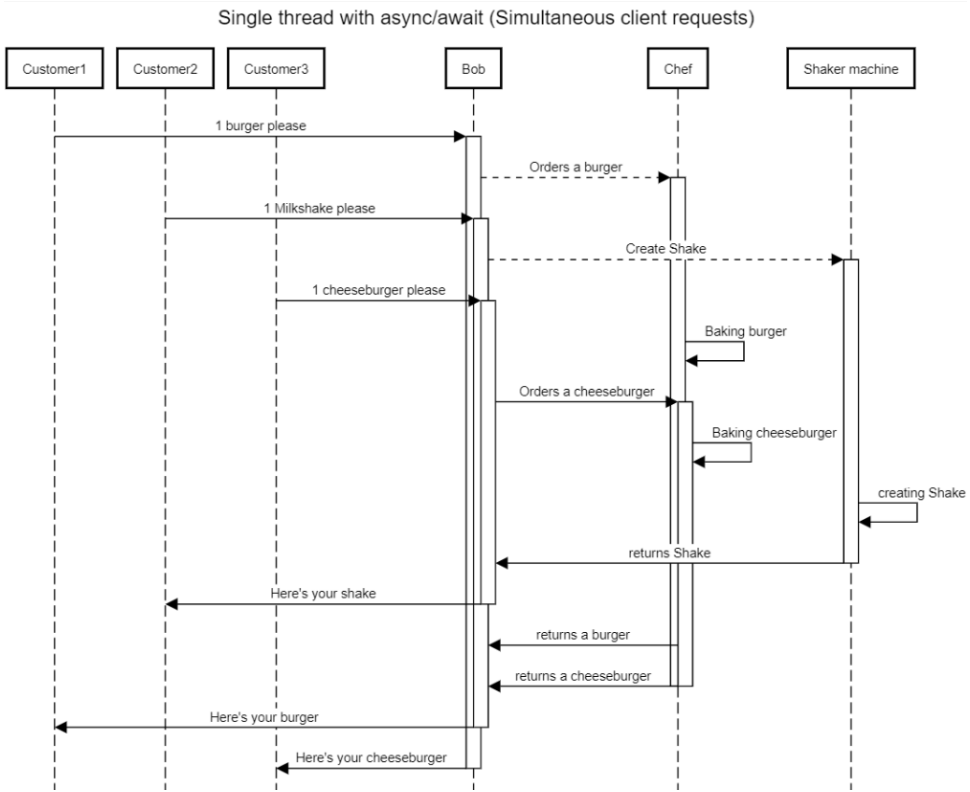
Step 4: async/await



Step 4: async/await



Step 4: async/await



Step 4: async / await

- Convert all classes which have (possible) **time consuming operations**
- EF supports using async/wait
  - Only required for **retrieving** data & **SaveChanges**
- Modify
  - Repository + interfaces
  - UnitOfWork
  - Services
  - Controllers

Step 4: async/await

```
2 references
public interface IGenericRepository<T>
{
    1 reference
    Task<IEnumerable<T>> GetAll(int pageNr, int pageSize);

    8 references
    Task<T> GetById(int id);
    2 references
    T Create(T newPerson);

    0 references
    T Update(T modifiedPerson);
    2 references
    void Delete(T person);
}
```

async

```
public interface IUnitofWork
{

    public IPeopleRepository PeopleRepository { get; }

    public IStoresRepository StoresRepository { get; }
    8 references
    Task Commit();
}
```

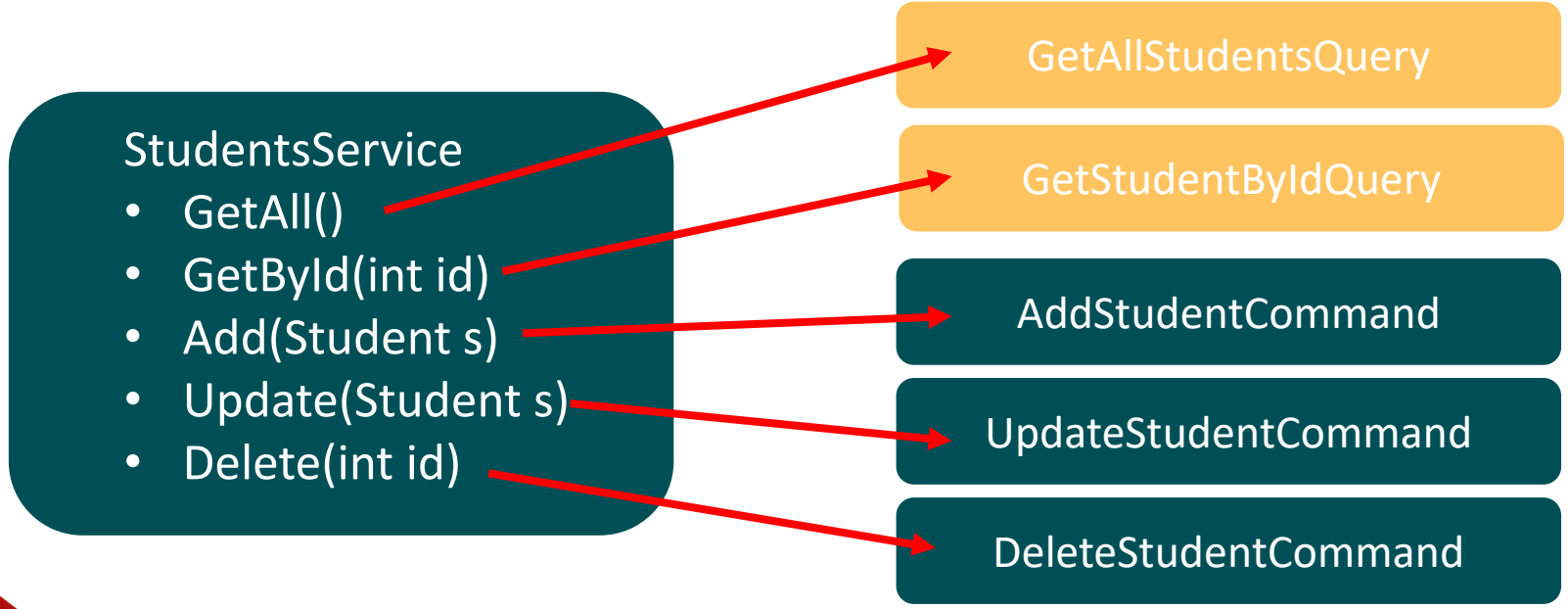
async

Step 5: CQRS & Mediator pattern

- Business logic & validations are now in the application core.
  - This can be done in services (eg. StudentService,..)
- Alternatively CQRS is implemented
  - = Command / Query Responsability Seggregation
  - Separation of
    - **Requesting** data (**Query**)
    - **Adding, Updating, Deleting** data (**Command**)

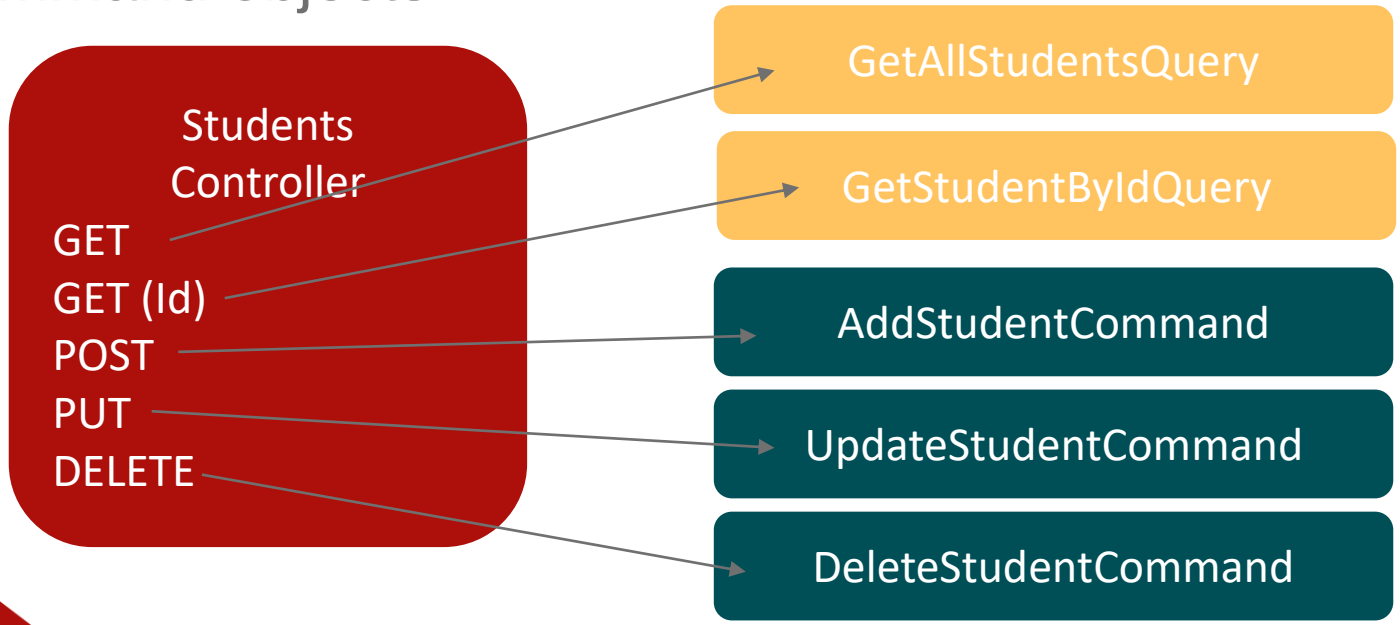
Step 5: CQRS & Mediator pattern

CQRS: Create separate Query and Command classes



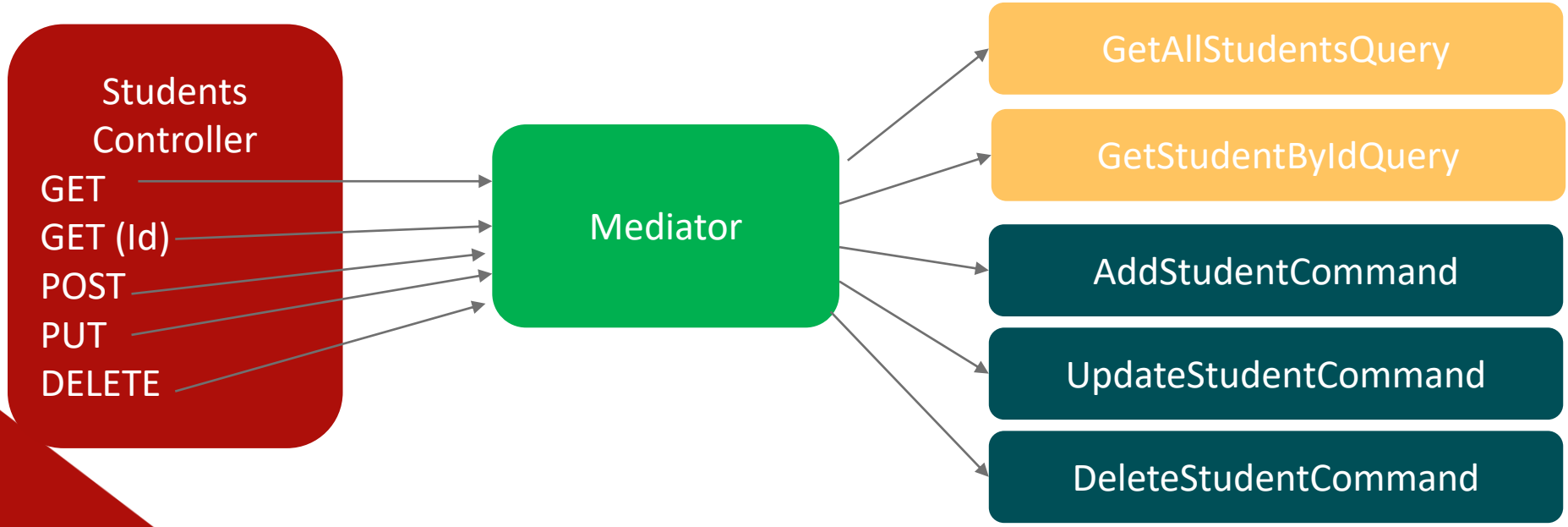
Step 5: CQRS & Mediator pattern

- This can lead to a very complex situation where each controller should have a reference to several query and command objects.



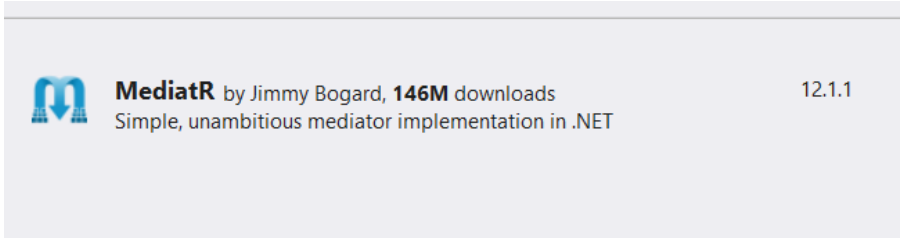
Step 5: CQRS & Mediator pattern

- A nice solution is to add the Mediator pattern



Step 5: CQRS & Mediator pattern

- Add **MediatR** package to Application project



- Initialize MediatR

```
0 references
public static class Registrator
{
    1 reference
    public static IServiceCollection RegisterApplication(this IServiceCollection services)
    {
        services.AddScoped<IPeopleService, PeopleService>();
        services.AddScoped<IStoresService, StoresService>();
        services.AddMediatR(cfg => cfg.RegisterServicesFromAssembly(Assembly.GetExecutingAssembly()));
        return services;
    }
```

Step 5: Convert services to Queries & Commands

```
namespace AP.MyGameStore.Application.CQRS.People
{
    3 references
    public class GetAllPeopleQuery : IRequest<IEnumerable<Person>>
    {
        2 references
        public int PageNumber { get; set; }
        2 references
        public int PageSize { get; set; }
    }

    1 reference
    public class GetAllPeopleQueryHandler : IRequestHandler<GetAllPeopleQuery, IEnumerable<Person>>
    {
        private readonly IUnitofWork uow;

        0 references
        public GetAllPeopleQueryHandler(IUnitofWork uow)
        {
            this.uow = uow;
        }
        0 references
        public async Task<IEnumerable<Person>> Handle(GetAllPeopleQuery request, CancellationToken cancellationToken)
        {
            return await uow.PeopleRepository.GetAll(request.PageNumber, request.PageSize);
        }
    }
}
```

```
2 references
public class PeopleService : IPeopleService
{
    private readonly IUnitofWork uow;

    0 references
    public PeopleService(IUnitofWork uow)
    {
        this.uow = uow;
    }

    1 reference
    public async Task<IEnumerable<Person>> GetAll(int pageNr, int page
    {
        return await uow.PeopleRepository.GetAll(pageNr, pageSize);
    }

    2 references
```

Step 5: Inject & use mediator in controller

```
0 references
public PeopleController(IPeopleService peopleService, IMediator mediator)
{
    this.peopleService = peopleService;
    this.mediator = mediator;
}
```

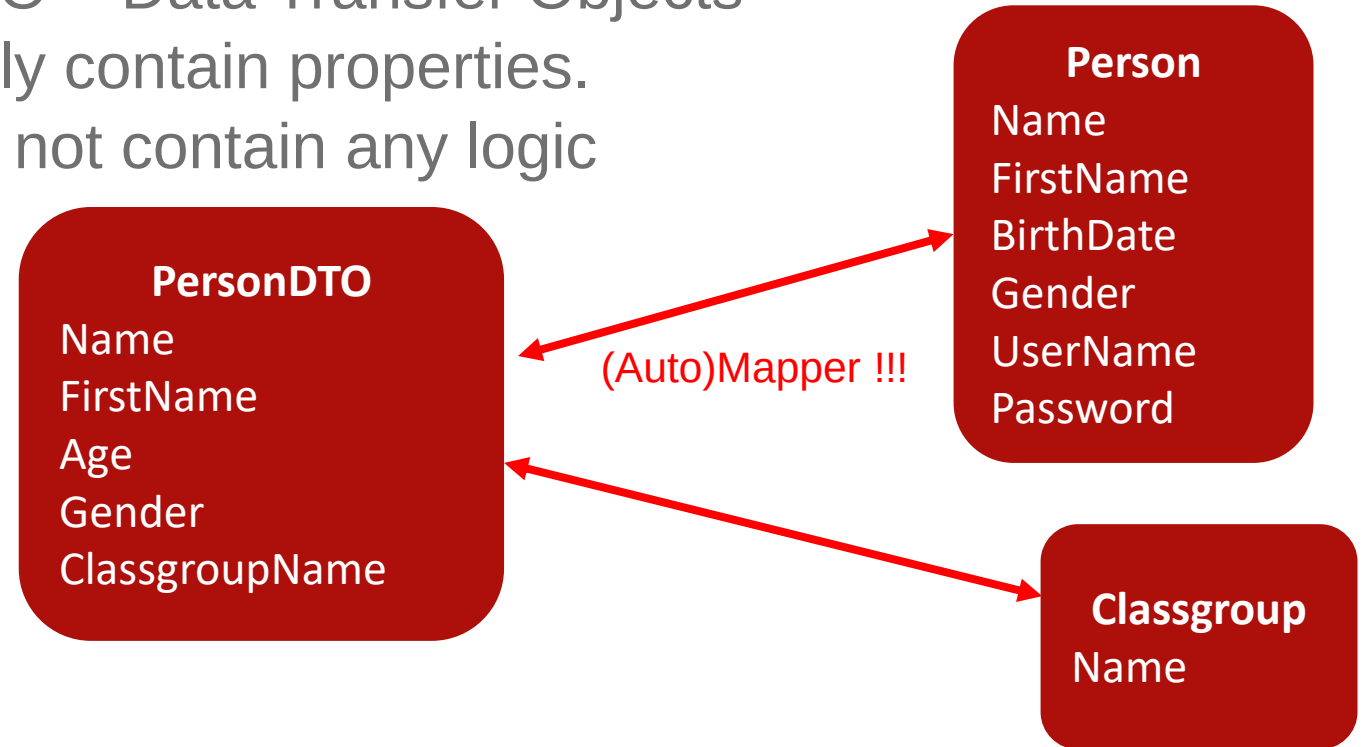
```
[HttpGet]   //api/people?lastname=Janssens
0 references
public async Task<IActionResult> GetAllPeople([FromQuery] string lastName, [FromQuery] int pageNr = 1, [FromQuery]
{
    //return Ok(await peopleService.GetAll(pageNr, pageSize));
    return Ok(await mediator.Send(new GetAllPeopleQuery() { PageNumber = pageNr, PageSize = pageSize }));
}

[HttpGet]
```

Step 6: Viewmodels and mapper

- Often we do not want to send complete entities in the response.
  - Not all properties can be publicly accessible (eg. Username, password,..)
  - Not all properties are relevant for the client
  - A **Get All** will most likely provide less info then **Get By Id**
  - The properties for Queries will often be different from the properties from Commands (see also CQRS)
  - …

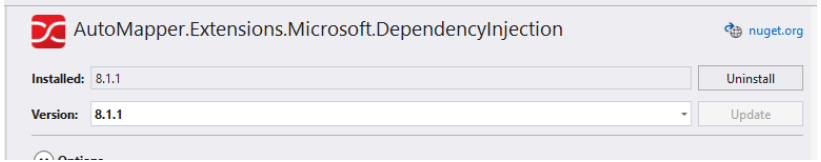
Step 6: Viewmodels and mapper

- For this purpose **Viewmodels / DTO's** are used
  - DTO = Data Transfer Objects
  - Only contain properties.
  - Do not contain any logic



Step6: DTOs and mapper

- Add Automapper to the ApplicationCore
  - This package will also install the Automapper package



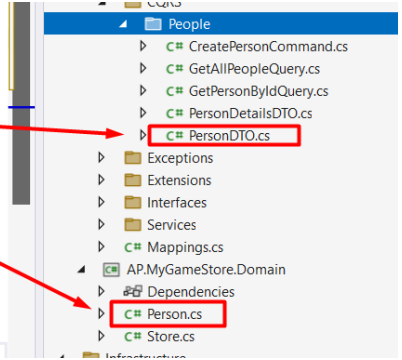
  - Initialize Automapper (D.I,...) during startup

```
namespace ApplicationCore
{
    0 references
    public static class ServiceExtensions
    {
        1 reference
        public static void AddApplicationCore(this IServiceCollection services)
        {
            services.AddAutoMapper(Assembly.GetExecutingAssembly());
            services.AddMediatR(Assembly.GetExecutingAssembly());
        }
    }
}
```

Step 6: DTOs & mapper

- Add DTO class with each Query / Command

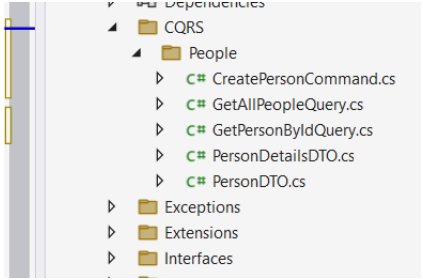
```
namespace AP.MyGameStore.Application.CQRS.People;

5 references
public class PersonDTO
{
    0 references
    public int Id { get; set; }
    0 references
    public string LastName { get; set; }
    0 references
    public string FirstName { get; set; }
    0 references
    public Gender Gender { get; set; }
}
```



- Define mapping for Automapper for each DTO

```
1 reference
public class Mappings : Profile
{
    0 references
    public Mappings()
    {
        CreateMap<Person, PersonDTO>();
        CreateMap<Person, PersonDetailsDTO>();
    }
}
```

1-way mapping



Step 6: DTOs & mapper

- Perform the mapping in the Query / Command

```
3 references
public class GetAllPeopleQuery : IRequest<IEnumerable<PersonDTO>>
{
    1 reference
    public int pageNr { get; set; } = 1;
    1 reference
    public int pageSize { get; set; } = 10;
}

1 reference
public class GetAllPeopleQueryHandler : IRequestHandler<GetAllPeopleQuery, IEnumerable<PersonDTO>>
{
    private readonly IUnitofWork uow;
    private readonly IMapper mapper;

    0 references
    public GetAllPeopleQueryHandler(IUnitofWork uow, IMapper mapper)
    {
        this.uow = uow;
        this.mapper = mapper;
    }
    0 references
    public async Task<IEnumerable<PersonDTO>> Handle(GetAllPeopleQuery request, CancellationToken cancellationToken)
    {
        return mapper.Map<PersonDTO[]>(await uow.PeopleRepository.GetAll(request.pageNr, request.pageSize));
    }
}
```

AP HOGESCHOOL ANTWERPEN

Step 6: DTOs & mapper

- More complex mappings are ofcourse possible
  - For example:

```
1 reference
public class Mappings : Profile
{
    0 references
    public Mappings()
    {
        CreateMap<Student, GetAllStudentsVM>()
            .ForMember(vm => vm.LastName, vm => vm.MapFrom(s => s.Name))
            .ForMember(vm => vm.AgeInDays, vm => vm.MapFrom(s => DateTime.Now.Subtract(s.Birth).TotalDays));
    }
}
```

Step 7: Status codes

- In this scenario only 2 status codes
  - NotFound (404)
  - OK (200)

```
[Route("{id}")]
[HttpPut]
0 references
public async Task<IActionResult> UpdateStudent(int id, [FromBody] AddStudentVM student)
{
    var currentStudent = await mediator.Send(new UpdateStudentCommand() { Id = id, Student = student });
    if (currentStudent == null)
        return NotFound();

    return Ok(currentStudent);
}
```

- Additionally there will be other business checks
  - Name too long
  - Age too low
  - …

Step 7: Status codes

- The business logic will make use of **exceptions**

```
0 references
public async Task<GetStudentVM> Handle(UpdateStudentCommand request, CancellationToken cancellationToken)
{
    var currentStudent = await repo.Get(request.Id);
    if (currentStudent == null)
        throw new KeyNotFoundException("the specified student was not found");

    //TODO: eventuele validatie (bv. BirthDate <= 1/1/2001,...)
    if (request.Student.LastName.Length > 50)
        throw new ValidationException("The lastname max. lenght is 50");
    if (request.Student.Birth.Year > 2003)
        throw new ValidationException("The student must be at least 18");
    //etc...
    currentStudent.Name = request.Student.LastName;
    currentStudent.FirstName = request.Student.FirstName;
    currentStudent.Birth = request.Student.Birth;
    currentStudent.Email = request.Student.Email;

    await repo.Update(currentStudent);

    return(mapper.Map<GetStudentVM>(currentStudent));
}
```

Step 7: status codes

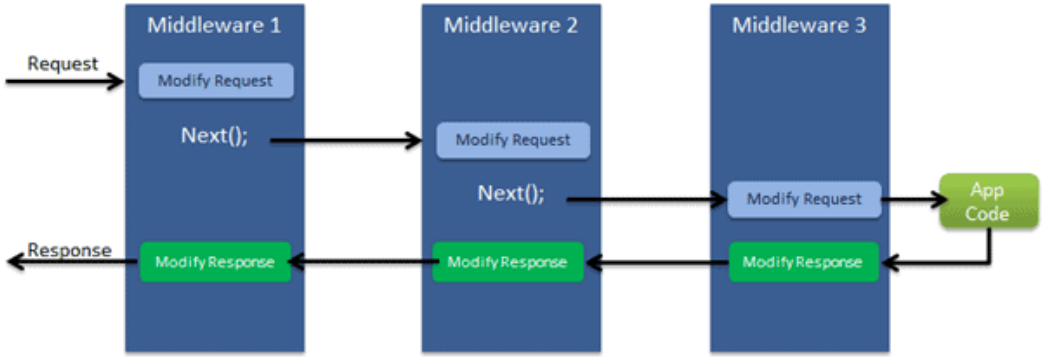
- These can be caught in the Controller

```
[Route("{id}")]
[HttpPut]
0 references
public async Task<IActionResult> UpdateStudent(int id, [FromBody] AddStudentVM student)
{
    try
    {
        return Ok(await mediator.Send(new UpdateStudentCommand() { Id = id, Student = student }));// repo.Get(id);
    }
    catch(Exception e)
    {
        switch (e)
        {
            case KeyNotFoundException:
                return NotFound(e.Message);
            case ValidationException:
                return BadRequest(e.Message);
            default:
                throw;
        }
    }
}
```

- Everywhere the same code in each controller/action…?
- Or…?

Step 7: status codes

- Let's take a look a the middleware ASP.NET core middleware pipeline.



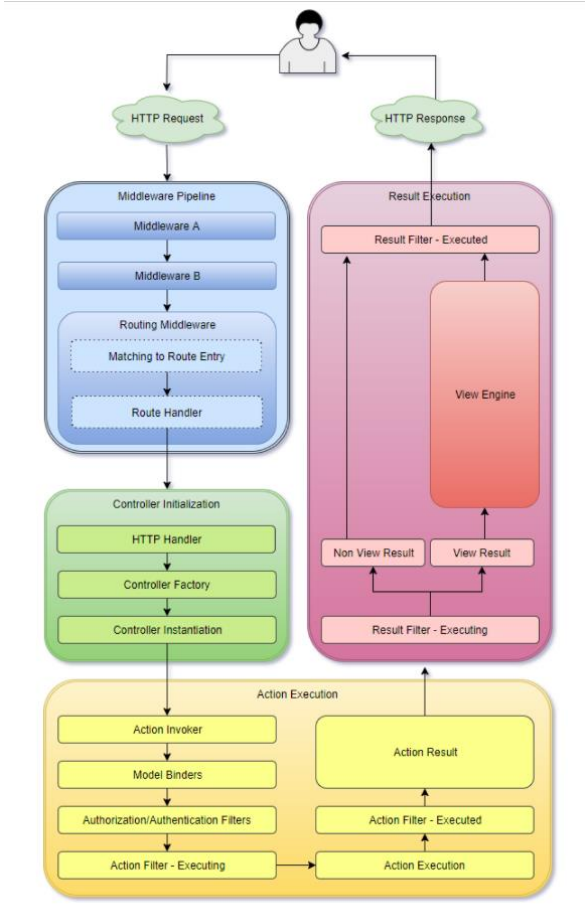
ASP.NET Core Middleware

- Each middleware is called by the previous middleware

Step 7: status codes

- The routing middleware:
  - Creates the controller
  - Invokes the action

- So what happens if we don't catch the exceptions inside the controllers….?



The ASP.NET Core MVC Pipeline

Step 7: status codes

- The exception will travel upwards until is caught either by:
  - Another middleware in the pipeline
  - The ASP framework (in that case it is converted to status code 500: internal server error)
  - During development a Developer “helper” page is shown

```
var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();

app.UseAuthorization();


app.MapControllers();

app.Run();
}
```

Step 7: status codes

- Empty middleware

```
public class OurOwnMiddleWare
{
    private readonly RequestDelegate _next;

    0 references
    public OurOwnMiddleWare(RequestDelegate next)
    {
        _next = next;
    }
    0 references
    public async Task InvokeAsync(HttpContext context)
    {
        //mw: doe uw ding...

        try
        {
            await _next(context);
        }
        catch(Exception e)
        {
            //mw: catch exceptions
        }
    }
}
```

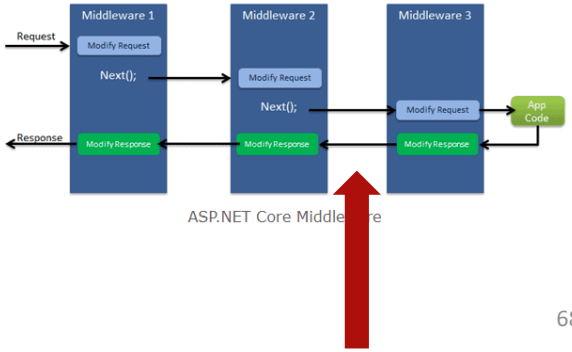
Step 7: status codes

- Add the middleware at the correct location in the pipeline

```
var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseMiddleware<OurOwnMiddelware>();

app.UseHttpsRedirection();

app.UseAuthorization();


app.MapControllers();

app.Run();
}
```

Step 7: status codes

- Solution:
  - We **do not** catch exceptions inside the **controllers**
  - But we add our own **ExceptionhandlingMiddleware** in between.
    - Catch there all the (possibly expected) **exceptions** from the Application Core.
    - And convert them into the corresponding **status** code
  - Write once… used everywhere !



Step 7: status codes

- Example of a ErrorHandler middleware

```
namespace AP.MyGameStore.WebAPI.Middleware;

2 references
public class ExceptionHandlingMiddleware
{
    private readonly RequestDelegate _next;

    0 references
    public ExceptionHandlingMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    0 references
    public async Task Invoke(HttpContext context)
    {
        try
        {
            await _next(context);
        }
        catch (Exception ex)
        {
            var response = new ErrorResponseInfo();
            response.Message = ex.Message;
            switch(ex)
            {
                case ValidationException:
                    response.StatusCode = StatusCodes.Status400BadRequest;
                    break;
                case RelationNotFoundException:
                    response.StatusCode = StatusCodes.Status404NotFound;
                    break;
            }
            context.Response.StatusCode = response.StatusCode;
            context.Response.ContentType = "application/json";
            await context.Response.WriteAsync(JsonSerializer.Serialize(response));
        }
    }
}
```

```
1 reference
public class ErrorResponseInfo
{
    3 references
    public int StatusCode { get; set; }
    1 reference
    public string Message { get; set; }
}
```

Step 7: status codes

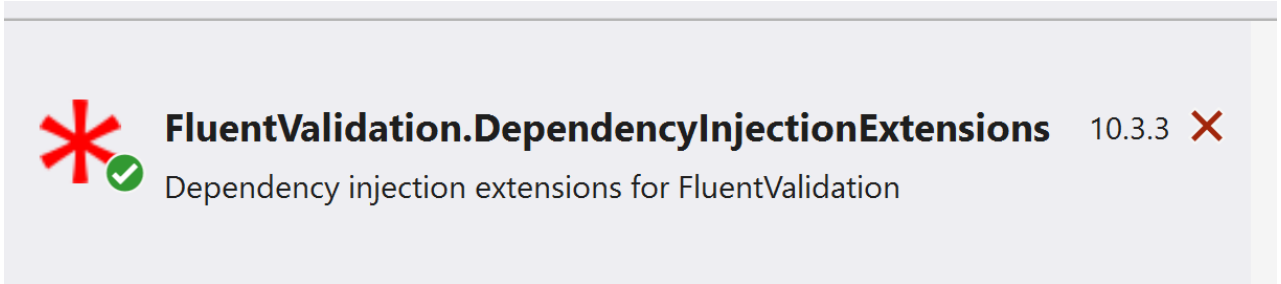
- Finishing touch with an extension method

```
0 references
public static class Registrator
{
    1 reference
    public static IApplicationBuilder UseErrorHandlingMiddleware(this IApplicationBuilder app)
    {
        app.UseMiddleware<ExceptionHandlingMiddleware>();
        return app;
    }
}
```

```
var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseErrorHandlingMiddleware();

app.UseHttpsRedirection();

app.UseAuthorization();


app.MapControllers();

app.Run();
}
```

Step 8: Validation

- In this architecture it is no longer possible to use the attributes on the “Model” for validation.
- We need a nice alternative for the Validation.
- Install the nuget Package:



Step 8: Validation

- Now we can create Validators.
- Typically a validator is created per command.

```
public class AddStudentCommandValidator : AbstractValidator<AddStudentCommand>
{
    0 references
    public AddStudentCommandValidator()
    {
        RuleFor(s => s.Student)
                .NotNull()
                .WithMessage("Student cannot be NULL");

        RuleFor(s => s.Student.Birth)
                .NotNull()
                .WithMessage("BirthDate cannot be NULL")
                .Must(b => b.Year < 2004)
                .WithMessage("Year of Birth must be smaller than 2004");

        RuleFor(s => s.Student.FirstName)
                .MaximumLength(15)
                .WithMessage("Firstname can be no more than 15 chars");
    }
```

AP HOGESCHOOL ANTWERPEN

Step 8: Validation

- Initialise the Validators in 1 call for D.I.

```
0 references
public static class ServiceExtensions
{
    1 reference
    public static void AddApplicationCore(this IServiceCollection services)
    {
        services.AddAutoMapper(Assembly.GetExecutingAssembly());
        services.AddMediatR(Assembly.GetExecutingAssembly());
        services.AddValidatorsFromAssembly(Assembly.GetExecutingAssembly());
    }
}
```

- After this we can inject a validator in the command and do the validation.
- Or alternativelly…

AP HOGESCHOOL ANTWERPEN

Step 8: Validation and Mediator

- We can also use the MediatR pipeline
- We can add validators in the pipeline
- The validator will be called automatically before the query / command !
- So we do not have to call the validator from each query / command.

Step 8: Validation and Mediator

- We do this by creating a ValidationBehaviour class

```
public class ValidationBehavior<TRequest, TResponse> : IPipelineBehavior<TRequest, TResponse>
    where TRequest : IRequest<TResponse>
{
    private readonly IEnumerable<IValidator<TRequest>> _validators;

    0 references
    public ValidationBehavior(IEnumerable<IValidator<TRequest>> validators)
    {
        _validators = validators;
    }

    0 references
    public async Task<TResponse> Handle(TRequest request, CancellationToken cancellationToken, RequestHandlerDelegate<TResponse> next)
    {
        if (_validators.Any())
        {
            var context = new FluentValidation.ValidationContext<TRequest>(request);
            var validationResults = await Task.WhenAll(_validators.Select(v => v.ValidateAsync(context, cancellationToken)));
            var failures = validationResults.SelectMany(r => r.Errors).Where(f => f != null).ToList();

            if (failures.Count != 0)
                throw new Exceptions.ValidationException(failures);
        }
        return await next();
    }
}
```

insert into the mediatR pipeline

collection of validators for the query/command

let them do the validation

throw exception upon any validation error

call next element in the pipeline

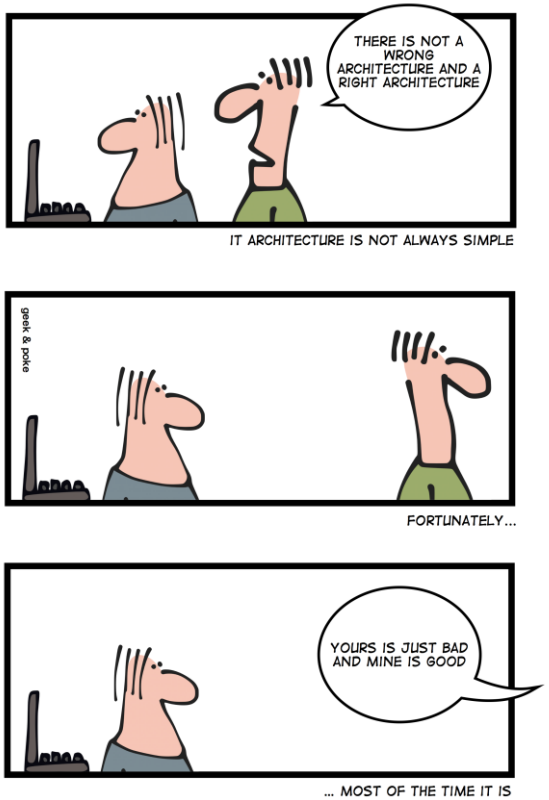
Step 8: Validation and Mediator

- Last step: initialisation of the ValidationBehaviour in the MediatR pipeline

```
public static class ServiceExtensions
{
    1 reference
    public static void AddApplicationLayer(this IServiceCollection services)
    {
        services.AddAutoMapper(Assembly.GetExecutingAssembly());
        services.AddMediatR(Assembly.GetExecutingAssembly());
        services.AddValidatorsFromAssembly(Assembly.GetExecutingAssembly());
        services.AddTransient(typeof(IPipelineBehavior<,>), typeof(ValidationBehavior<,>));

    }
}
```

- Async/await: https://www.carlrippon.com/scalable-and-performant-asp-net-core-web-apis-asynchronous-operations/
- CQRS & MediatR in ASP.NET core: https://www.hosting.work/cqrs-mediatr-aspnet-core/

Tenslotte..


THERE IS NOT A WRONG ARCHITECTURE AND A RIGHT ARCHITECTURE
IT ARCHITECTURE IS NOT ALWAYS SIMPLE
geek & poke
FORTUNATELY...
YOURS IS JUST BAD AND MINE IS GOOD
... MOST OF THE TIME IT IS