PRESENTING…

TAMING THE CQRS PATTERN

By Paul Walter

Initialize the Code!

Clone the repository and read the README.md file.

It's a simplified example of most of the ideas we've gleaned from the Microsoft eShops project.

Note, this simplified project isn't addressing external messaging, just from within a microservice.

What is CQRS?

- It is a *software pattern* that promotes separating your reads from your writes.
- Promotes an event-based programming model.
- Useful for microservices.
- It does this for performance reasons.
  - Write side = complex joins
  - Read side = no joins (if you can)
- The acronym stands for "Command and Query Responsibility Segregation."

Why does CQRS have “Wilds”?

- It’s COMPLEX.
- New programming paradigm.
- Not many well documented projects.
- There’s differing opinions implementation.
- Going into “the unknown” is a scary prospect.
- Differing flavors of CQRS: Event Sourcing vs. Not.
- Many times it’s coupled with Domain Driven Design (DDD) which also has a high learning curve.

Why use an Event Driven Model?

- It let's our applications respond to inner and outer events.
  - Inner Events = from within our Microservice.
  - Outer Events = msgs from other Microservices
- It allows us to orchestrate transactions with other Microservices or respond to events within our system.

NOW…A STORY ABOUT FOOD

Before we discuss CQRS more, let's look at a parallel story we all know…

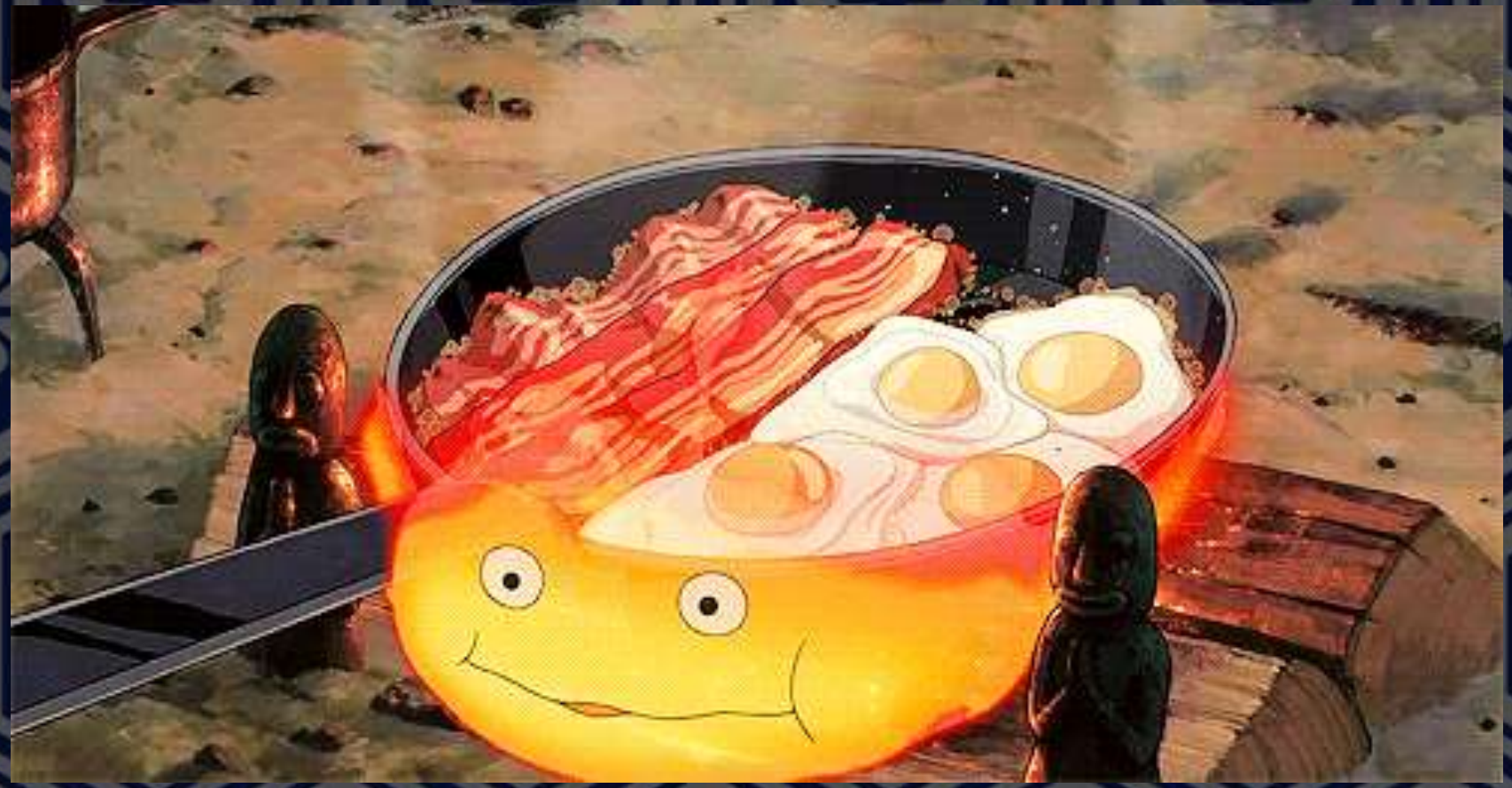

...WHICH TAKES PLACE IN A RESTAURANT
(YOU KNOW THIS STORY, YEAH?)


SO…THE *CLIENT* CONSULTS THE MENU
…TELLS THE WAITER WHAT THEY WOULD LIKE
…THEN THE *WAITER* CREATES THE *ORDER*
(The client is the domain expert on what will be delicious for them)

COOK RECEIVES THE ORDER

(Check them out in the top right. Those Orders are hanging out in a queue.)

. . .AND PREPARES THE MEAL



(nom nom nom)


WAITER SERVES THE MEAL
AND CREATES A BILL


CLIENT EATS THEIR MEAL!
And is delighted!

CLIENT CONSULTS THEIR BILL

(Hopefully, there isn't any Japanese profanity on this)

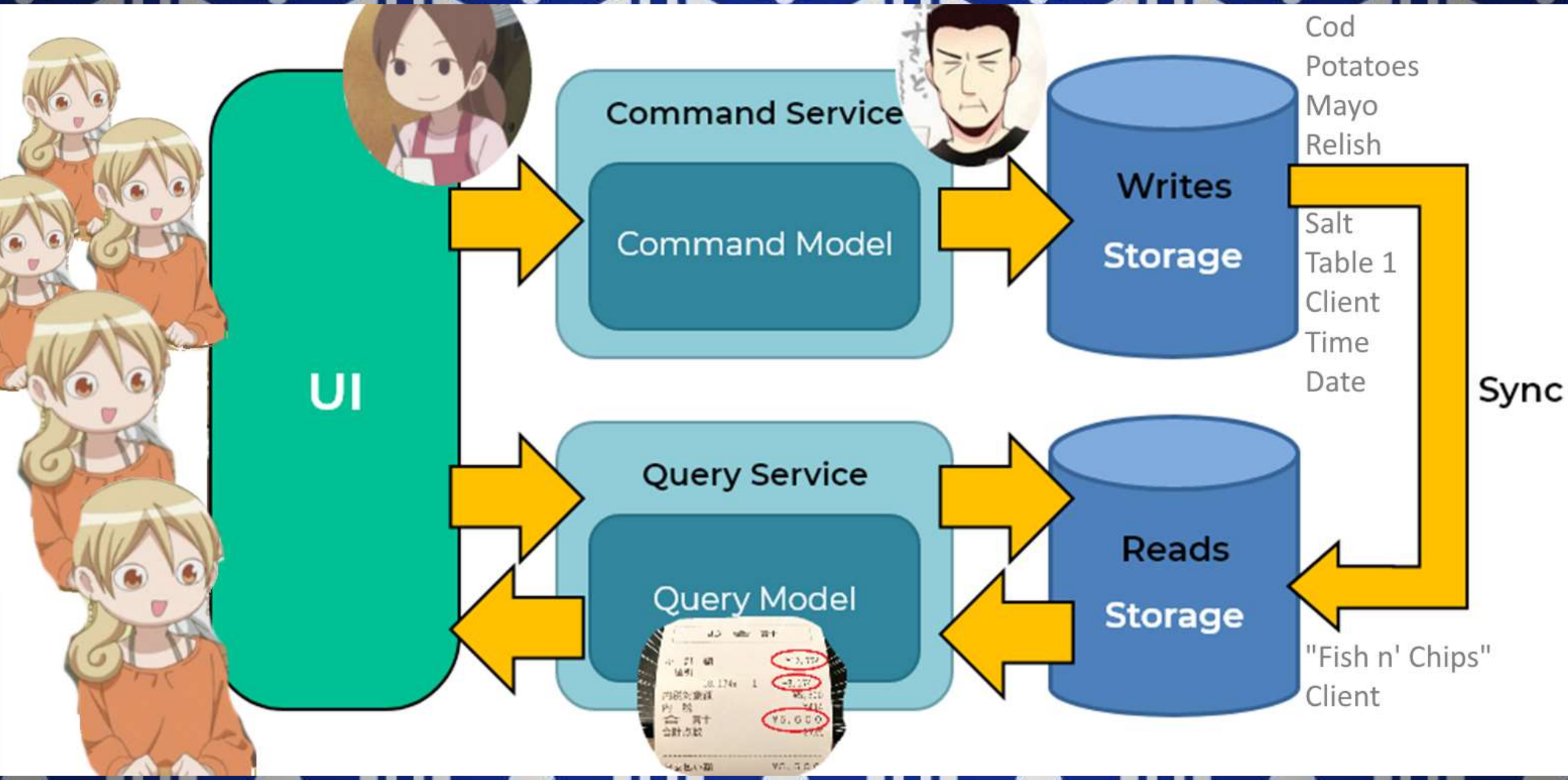

LASTLY…
CLIENT PAYS
THEIR BILL

THE END

(See? You knew that story!)

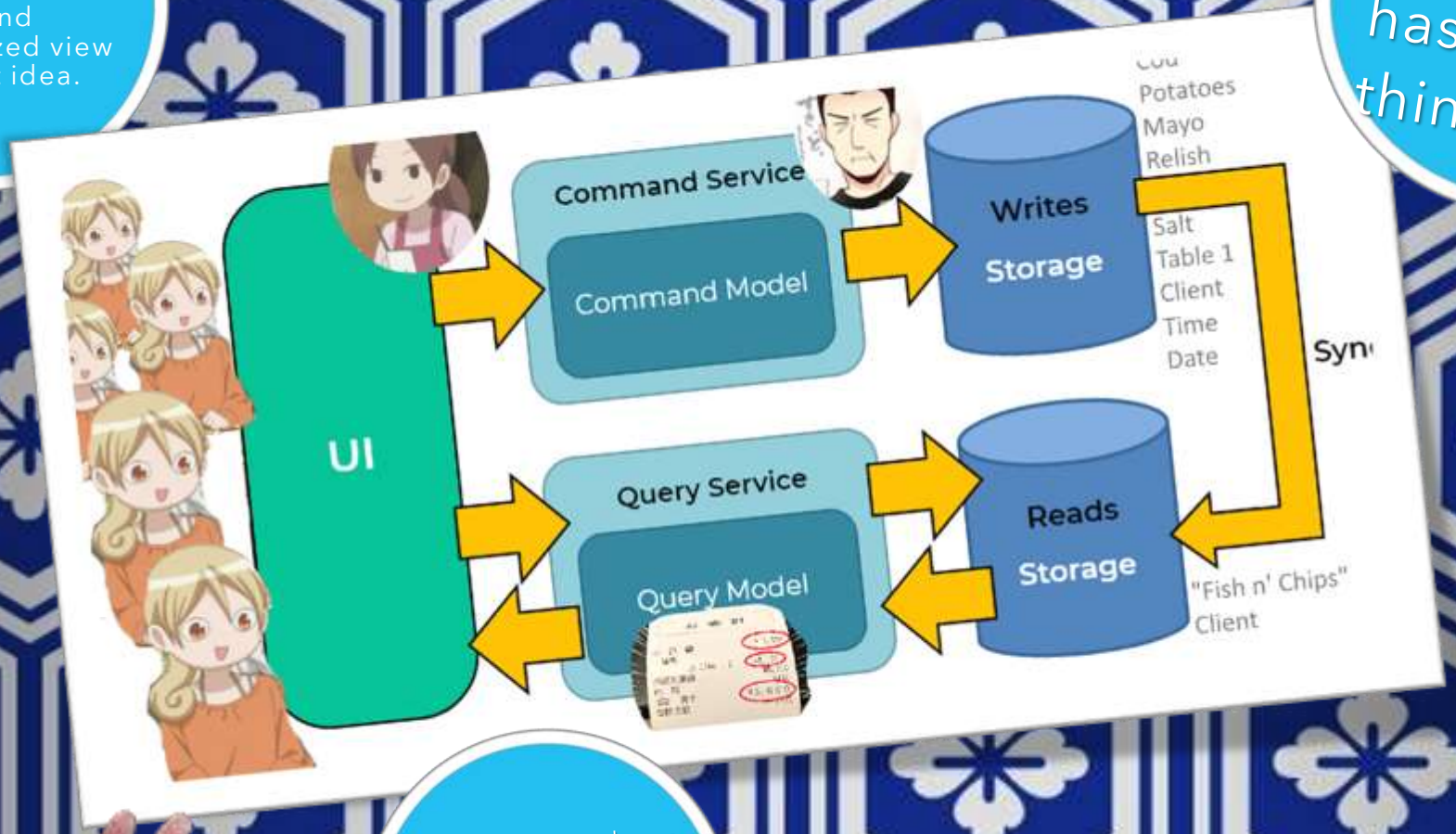
CQRS

...is a similar story


Command Service
Command Model
Writes
Storage
Cod
Potatoes
Mayo
Relish
Salt
Table 1
Client
Time
Date
Sync
UI
Query Service
Query Model
Reads
Storage
"Fish n' Chips"
Client

See all the users? There will be more reads than writes so a cached and summarized view is a great idea.

The "Writes" Database has a lot of things, no?



Query service has a "***denormalizer***" concept that flattens objects so that any collections become flat properties

Look how little there is in the "Reads" database!!!!

QUESTION 1

...how do they get what they need to do their job?

AUTOFAC

(IOC)

(TOTORO MAGICALLY SUPPLYING FUN)

AUTOFAC

- Autofac is a *magic* Inversion of Control Container (IOC).
- *IOC Container* is a framework that injects dependencies through a constructor.
  - Uses the Dependency Inversion Principle (program to an abstraction)
  - Dependency Inversion Principle is part of the SOLID

AUTOFAC
Configuration

```
// ConfigureContainer is where you can register things directly
// with Autofac. This runs after ConfigureServices so the things
// here will override registrations made in ConfigureServices.
// Don't build the container; that gets done for you by the factory.
0 references | paul_walter, 17 hours ago | 1 author, 3 changes
public void ConfigureContainer(ContainerBuilder builder)
{
    new AutofacStart(Configuration, builder);
}
```

Startup.cs

```
// Entity Framework Repository
builder.RegisterType<PetRepository>()
    .As<IPetRepository>()
    .InstancePerLifetimeScope();

// CQRS Queries Repository.
builder.RegisterType<PetStoreQueriesRepository>()
    .As<IPetStoreQueriesRepository>()
    .InstancePerLifetimeScope();

// Entity Framework Repository
builder.RegisterType<SecretsManager>()
    .As<ISecretsManager>()
    .InstancePerLifetimeScope();
```

AutofacApplicationModule.cs

See the PetStoreQueriesRepository?

AUTOFAC

Usage: After it's configured, it's magic: Just identify which dependencies you'd like, like so….see Macho Man pointing to the "IPetStoreQueriesRepository"? (Ohhhh yeah!)

```
/// <summary>
/// This constructor is for Autofac
/// </summary>
/// <param name="mediator"></param>
/// <param name="logger"></param>
0 references | paul_walter, 15 hours ago | 1 author, 2 changes
public PetStoreQueryController(IMediator mediator, ILogger logger, IPetStoreQueriesRepository petQ                ) : base()
{
    _mediator = mediator;
    _logger = logger;
    _petQueriesRepo = petQueriesRepo;

    // initialize the dictionaries for big-0 time savings
    _petTypeDictionary = EnumUtils.CreateDictionaryByToString<PetTypeValue>();
    _petSortDictionary = EnumUtils.CreateDictionaryByToString<PetSortValue>();
```

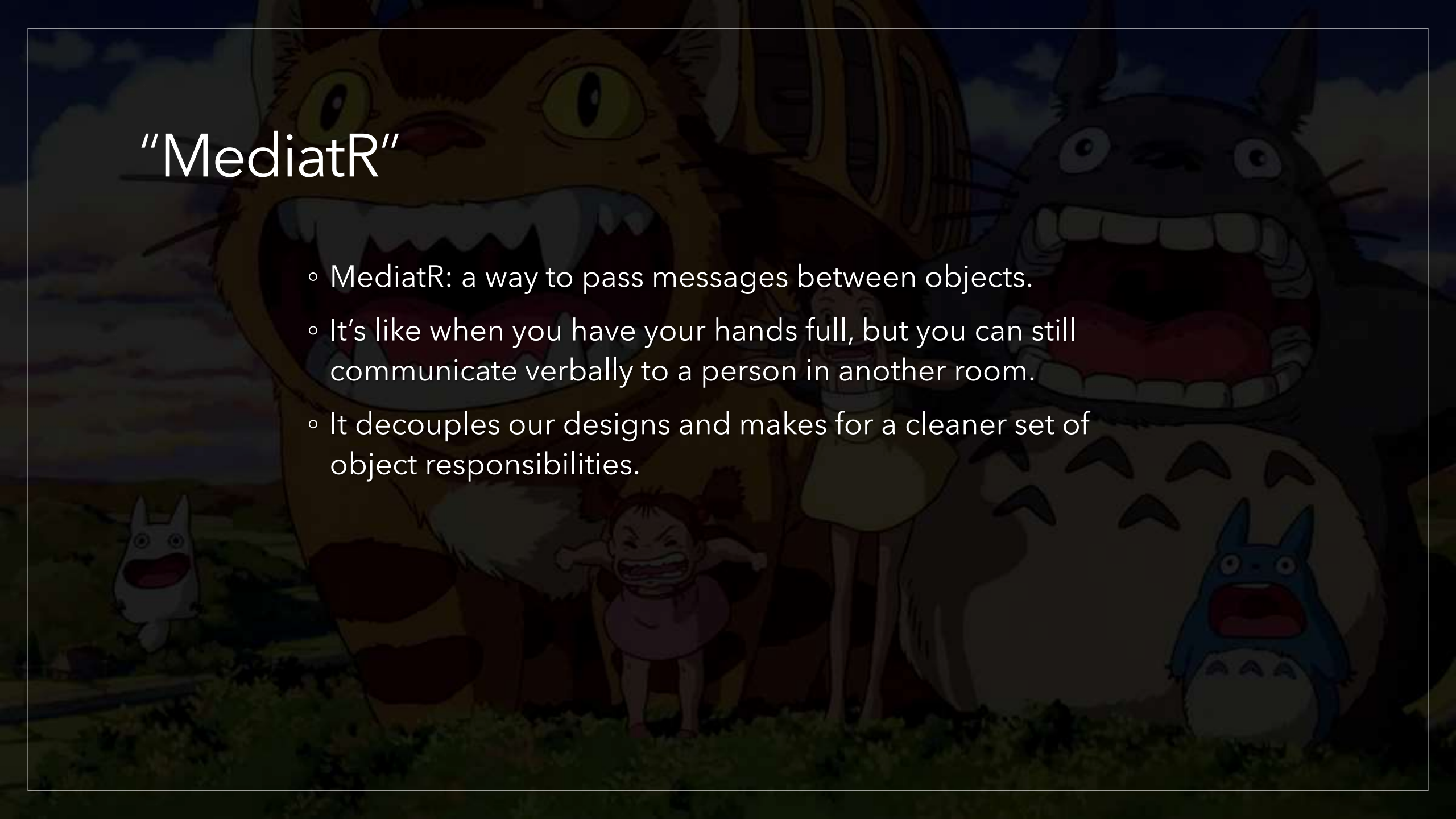
PetStoreQueryController.cs

QUESTION 2

...how do they communicate internally?

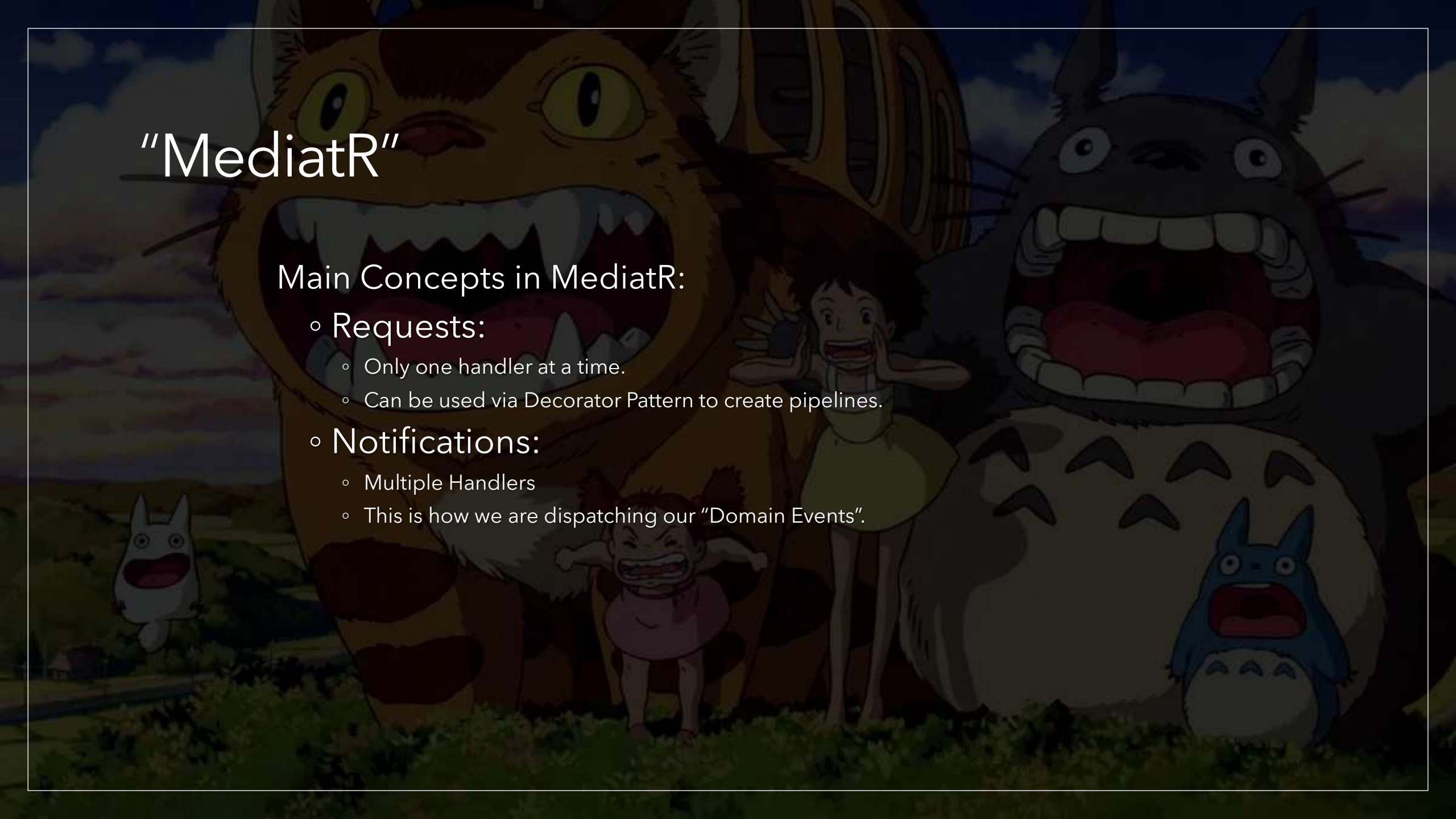
(Externally is another presentation, aka msg'ing between microservices)


"MediatR"



“MediatR”

- MediatR: a way to pass messages between objects.
- It’s like when you have your hands full, but you can still communicate verbally to a person in another room.
- It decouples our designs and makes for a cleaner set of object responsibilities.

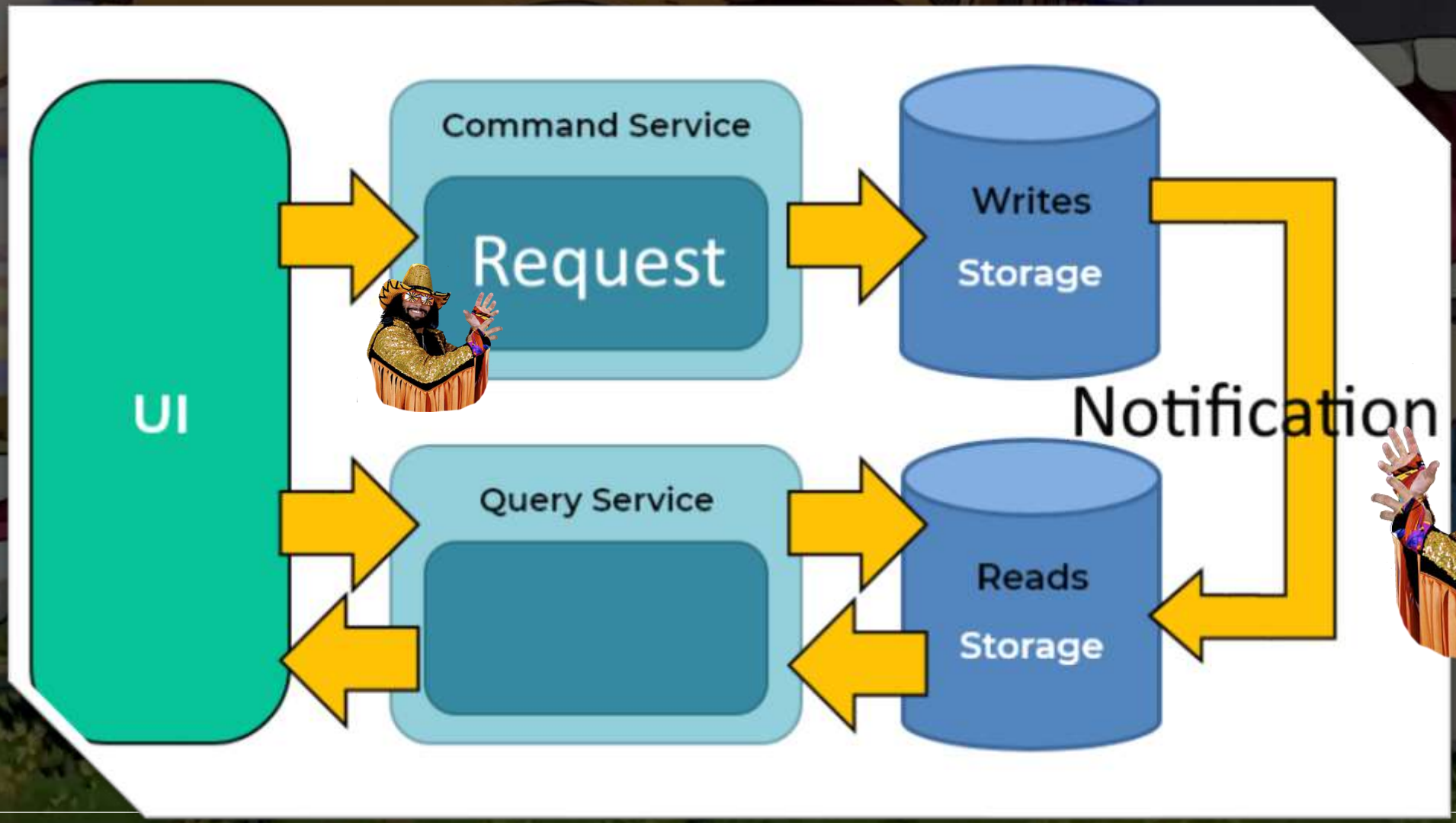


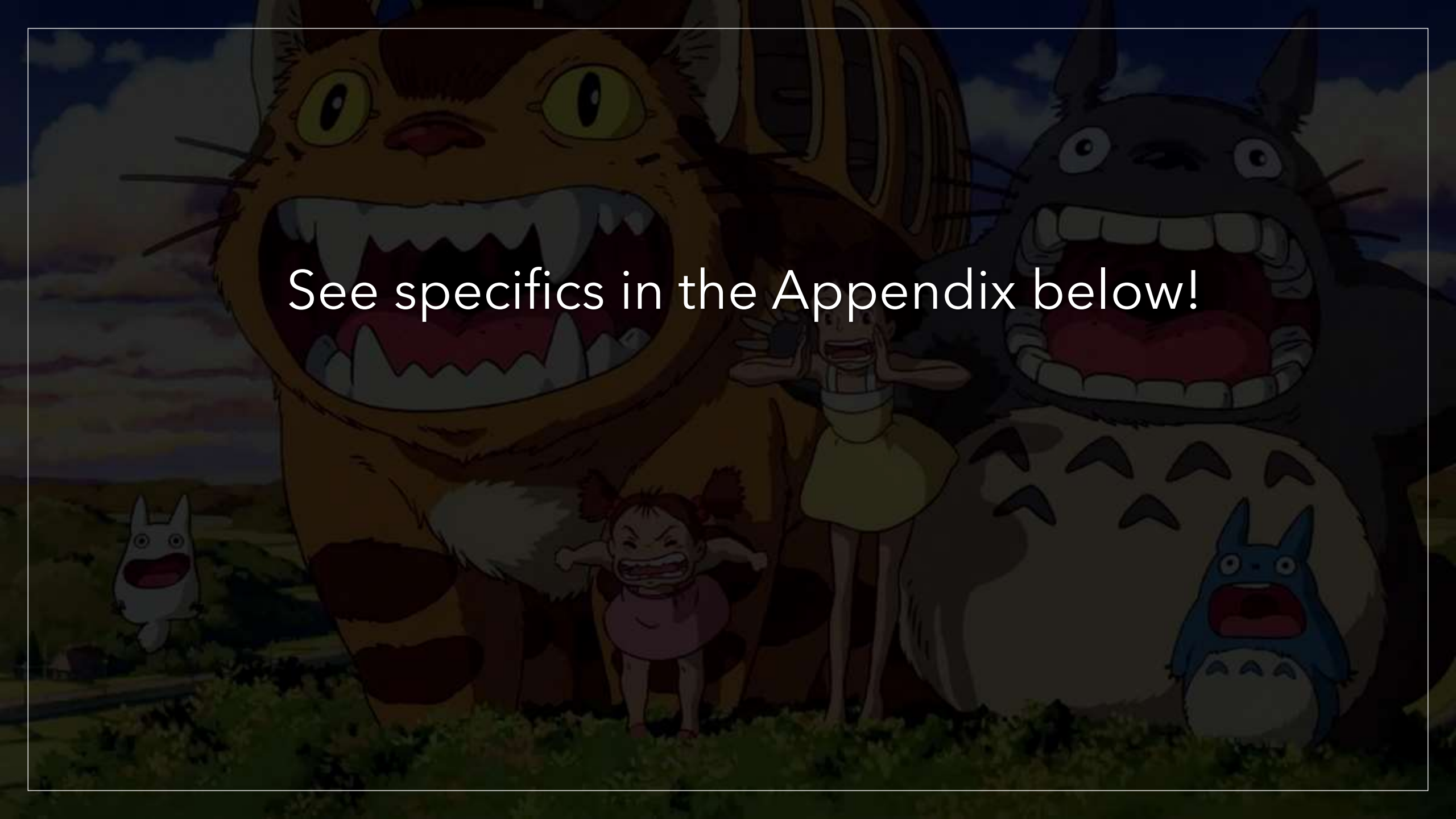
“MediatR”

Main Concepts in MediatR:

- Requests:
  - Only one handler at a time.
  - Can be used via Decorator Pattern to create pipelines.
- Notifications:
  - Multiple Handlers
  - This is how we are dispatching our “Domain Events”.

"MediatR"


Command Service
Request
Writes
Storage
UI
Notification
Query Service
Reads
Storage


See specifics in the Appendix below!

QUESTION 3

...how do validate business rules easily?

Fluent Validation

What is it?

- It's a library for writing object validations.

Why use it?

- Emily and I think it's great!
- Well, that and it's fast to ensure *complex* business are correct &, if not, send the correct errors.
- Notice my *errors*…I'm referencing Error Codes defined in my Open API yaml file!

Where is it?

- Check out the *CreatePetValidator.cs* file

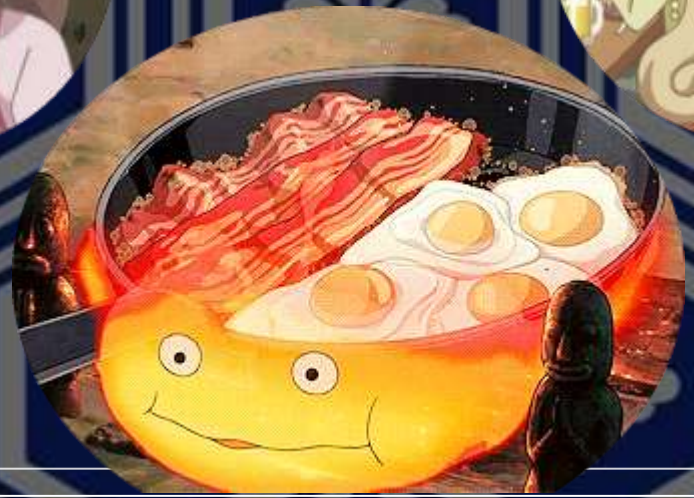
```
2 references | paul_walter, 1 day ago | 1 author, 3 changes
public class CreatePetValidator : AbstractValidator<CreatePetCommand>
{
    0 references | paul_walter, 1 day ago | 1 author, 3 changes
    public CreatePetValidator()
    {
        // ================================
        //       TOP LEVEL MEMBERS
        // ================================

        // Make sure it's an empty guiud
        RuleFor(cmd => cmd.Pet.ResourceID)
            .Equal(new Guid())
            .WithErrorCode(PetStoreErrorValue.Pet_Resource_ID_must_be_000000000000

        // Make sure name isn't empty
        RuleFor(cmd => cmd.Pet.Name)
            .NotEqual(string.Empty)
            .WithErrorCode(PetStoreErrorValue.Pet_Name_is_required.ToString());

        // make sure type isn't empty
        RuleFor(cmd => cmd.Pet.Type)
            .NotNull()
            .WithErrorCode(PetStoreErrorValue.Pet_Name_is_required.ToString());
```

QUESTION 4



...how do we scaffold this app?

OpenAPI

What is it?

It's technology for enforcing API contracts.

Why use it?

It promotes a high level, design-first, type of thinking.

You can clearly communicate your ideas across development teams.

You can *scaffold the heck outta it*!!!!

Where is it?

Check out the `README.md` in the `PetStore.OpenAPI` project

OpenAPI

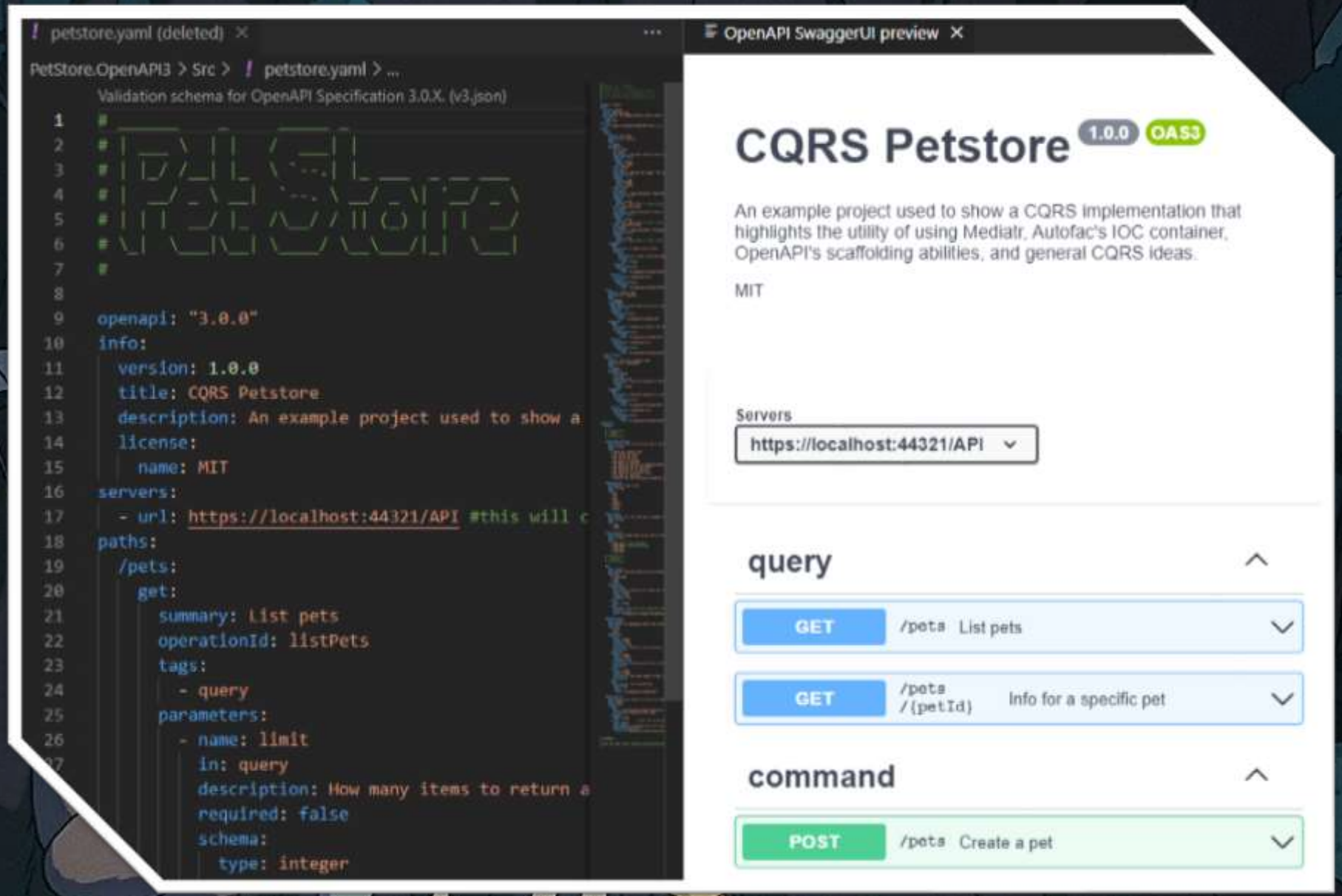
Editing / Testing

I like to use VS Code to edit & test my OpenAPI files.

There are two flavors: JSON and Yaml. Yaml is easier to read imo.

There's also a good plugin called "OpenAPI (Swagger) Editor" by 42Crunch.

You can also import OpenAPI files into Postman



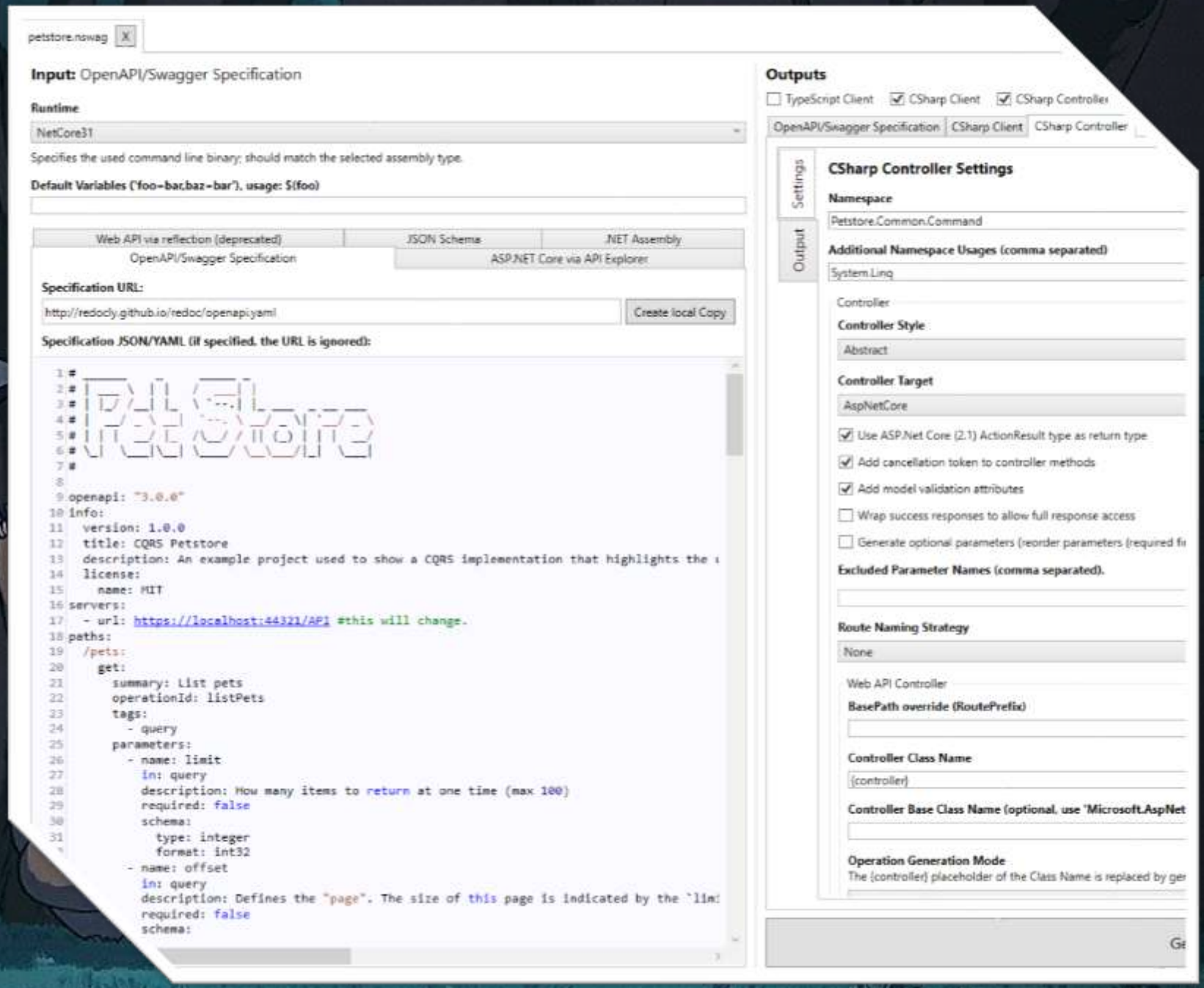
OpenAPI Scaffolding

There are many different scaffolders depending on what computer language you are using.

For C#, Nswag is my preferred scaffolder. It uses a GUI called "Nswag Studio".

You can scaffold C# servers, C# & JavaScript clients.

Check out the Microsoft Tutorial for more information.

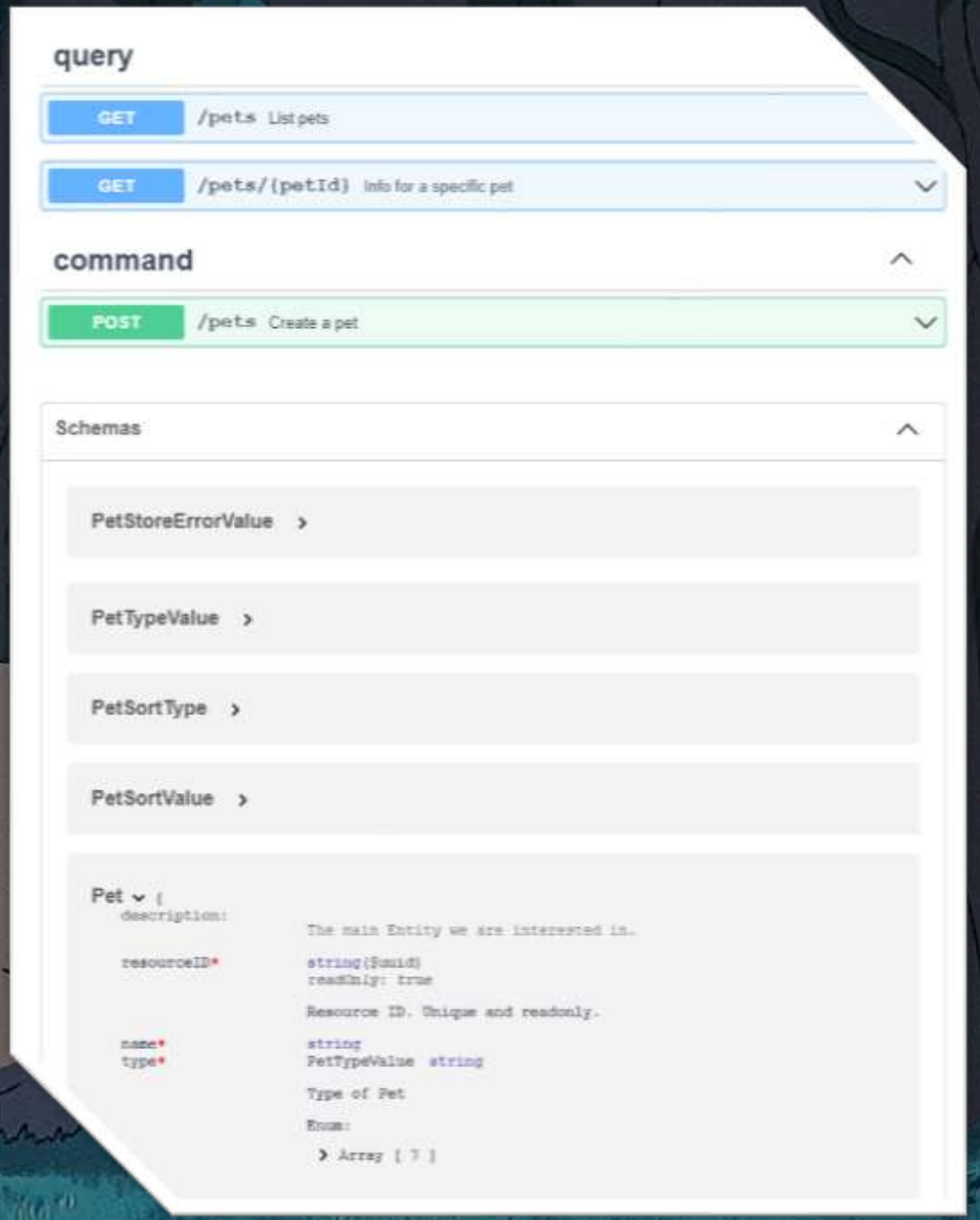


OpenAPI
Existing Code

You can also hook Nswag into your applications to create a OpenAPI document for you.

This provides great visibility on what your API is doing.

By hooking in Nswag, it can also generate a Swagger Editor view so you can inspect your endpoints AND all the objects being serialized.



DEMO

Let’s see this baby this in action!

(Please clone my CQRS PetStore project.)

Final Words

- *Encapsulate* your IOC configurations so you can also use them in your Test projects.
- Dotnet 3+ has its own IOC Container, Autofac isn't the only one.
- If we move to a container style (Docker is good, but Kubernetes looks better) of development we can also spin up external event brokers and also coordinate external events.
  - RabbitMQ article: https://codeburst.io/get-started-with-rabbitmq-on-docker-4428d7f6e46b
  - Promising Github project: https://github.com/changhuixu/rabbitmq-labs

APPENDIX

Technology

Autofac: https://autofac.org/
Fluent Validation: https://fluentvalidation.net/
Martin Fowler: https://martinfowler.com/bliki/CQRS.html
MediatR: https://github.com/jbogard/MediatR
Microsoft eShops: https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/introduce-eshoponcontainers-reference-app
Source Making (software patterns): https://sourcemaking.com/design_patterns

Assets

Shin-chan: https://en.wikipedia.org/wiki/Crayon_Shin-chan
Ghibli : https://ghiblicollection.com/
Wakako Zake: https://en.wikipedia.org/wiki/Wakakozake

Appendix

"MediatR"

Request: Create the Request Object

Notice the `IRequest` interface is of type MediatR.IRequest

```
7 references | paul_walter, 1 day ago | 1 author, 1 change
public class CreatePetCommand : IRequest<Pet>
{
    5 references | paul_walter, 1 day a
    public Pet Pet { get; }

    1 reference | paul_walter, 1 day ago | 1 author, 1 change
    public CreatePetCommand(Pet pet)
    {
        Pet = pet;
    }
```

```
interface MediatR.IRequest<out TResponse>
Marker interface to represent a request with a response

TResponse is Pet
```

CreatePetCommand.cs

Appendix

"MediatR"

Request: Create the Request Handler

```
2 references | paul_walter, 22 hours ago | 1 author, 2 changes
public class CreatePetCommandHandler : IRequestHandler<CreatePetCommand, Pet>
{
    private readonly IPetRepository _petRepository;
    private readonly ILogger _logger;

    0 references | paul_walter, 1 day ago | 1 author, 1 change
    public CreatePetCommandHandler(IPetRepository petRepository, ILogger logger)
    {
        _petRepository = petRepository ?? throw new ArgumentNullException(nameof(petRepository));
        _logger = logger ?? throw new ArgumentNullException(nameof(logger));
    }

    0 references | paul_walter, 22 hours ago | 1 author, 2 changes
    public async Task<Pet> Handle(CreatePetCommand command, CancellationToken cancellationToken)
    {
        Pet pet = null;
        bool success = false;
        DomainModels.Pet newPet = null;
        DomainModels.Pet existingPet = null;
```

CreatePetCommandHandler.cs

Appendix

"MediatR"

Request: Configure with Autofac

```
//
// REGISTER COMMAND CLASSES (they implement IRequestHandler) in assembly holding the Commands
//
builder.RegisterAssemblyTypes(typeof(CreatePetCommand).GetTypeInfo().Assembly)
    .AsClosedTypesOf(typeof(IRequestHandler<,>));

//
// REGISTER COMMAND HANDLERS (IRequestHandler)
//
builder.RegisterAssemblyTypes(typeof(CreatePetCommandHandler).GetTypeInfo().Assembly)
  .AsClosedTypesOf(typeof(IRequestHandler<,>));
```

AutofacMediatorModule.cs

Appendix

"MediatR"

Request: Dispatch

```
[Microsoft.AspNetCore.Mvc.HttpPost, Microsoft.AspNetCore.Mvc.Route("pets")]
1 reference | paul_walter, 5 days ago | 1 author, 3 changes
public override async Task<ActionResult<Pet>> CreatePet([FromBody] Pet pet, CancellationToken cancellationToken = default)
{
    try
    {
        CreatePetCommand cmd = new CreatePetCommand(pet);
        Pet updatedPet = await _mediator.Send(cmd, cancellationToken);

        return Ok(updatedPet);
    }
    catch (PetStoreException exp)
    {
        return BadRequest(exp);
    }
    catch (Exception)
    {
        // it has already been logged, no need to re-log the exception
        return StatusCode((int)HttpStatusCode.InternalServerError);
```

PetStoreCommandController.cs

Appendix

"MediatR"

Notification: Create Notification Object

```
4 references | paul_walter, 5 days ago | 1 author, 1 change
public class PetStoreDomainEvent : INotification
{
    public readonly PetStoreEventDTO PetStoreDTO;

    1 reference | paul_walter, 5 days ago | 1 author, 1 change
    public PetStoreDomainEvent(
        Guid resourceID,
        string name,
        string type)
    {
        PetStoreDTO = new PetStoreEventDTO(
            resourceID,
            name,
            type);
    }
```

- Note that our Domain Events are all notifications.
- In the Domain layer we add these events to a Domain object
- Then, when we pass them to the Infrastructure layer, if everything goes ok, we dispatch the Domain event.

PetStoreDomainEvent.cs

Appendix

“MediatR”

Notification: Autofac Configuration

```
//
// REGISTER DOMAIN EVENT HANDLERS (they implement INotificationHandler<>) in assembly holding the Domain Events
//
builder.RegisterAssemblyTypes(typeof(CreatePetDomainEventHandler).GetTypeInfo().Assembly)
    .AsClosedTypesOf(typeof(INotificationHandler<>));
```

AutofacMediatorModule.cs

Appendix

"MediatR"

Pipeline: Logging, Validations and Transactions

- So in addition to Requests, you can chain them together so that every Request has to proceed through a series of handlers. In the PetStore example, I've got a Logging & Validation handlers set up.
- See the eShops github example for more context (See Appendix)
- This is how I'm configuring Autofac to wire up Logging & Validations.

```
// finally register our custom code (individually, or via assembly scanning)
// - requests & handlers as transient, i.e. InstancePerDependency()
// - pre/post-processors as scoped/per-request, i.e. InstancePerLifetimeScope()
// - behaviors as transient, i.e. InstancePerDependency()
builder.RegisterGeneric(typeof(LoggingBehavior<,>))
    .As(typeof(IPipelineBehavior<,>));

// TODO: Uncomment this line when we want to do Command Validation BEFORE it gets to the domain layer
//
builder.RegisterGeneric(typeof(PetStoreValidatorPipelineBehavior<,>))
    .As(typeof(IPipelineBehavior<,>));
```

AutofacStart.cs