



PRESENTING:

"TAMING THE CQRS PATTERN"

By Paul Walter



Download the Code!

Clone or download the code repository. Execute the migration script in "*PetStore.Infrastructure/Scripts/Migrations*" in your SQL Server DB.

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...

AND 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

小 計 額
値引

¥13,774

88,174x 1

-8,174

内税対象額

¥5,600

内 税

¥414

合 計

¥5,600

合計点数

27点

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

お支払い額

¥5,600



LASTLY...
CLIENT PAYS
THEIR BILL



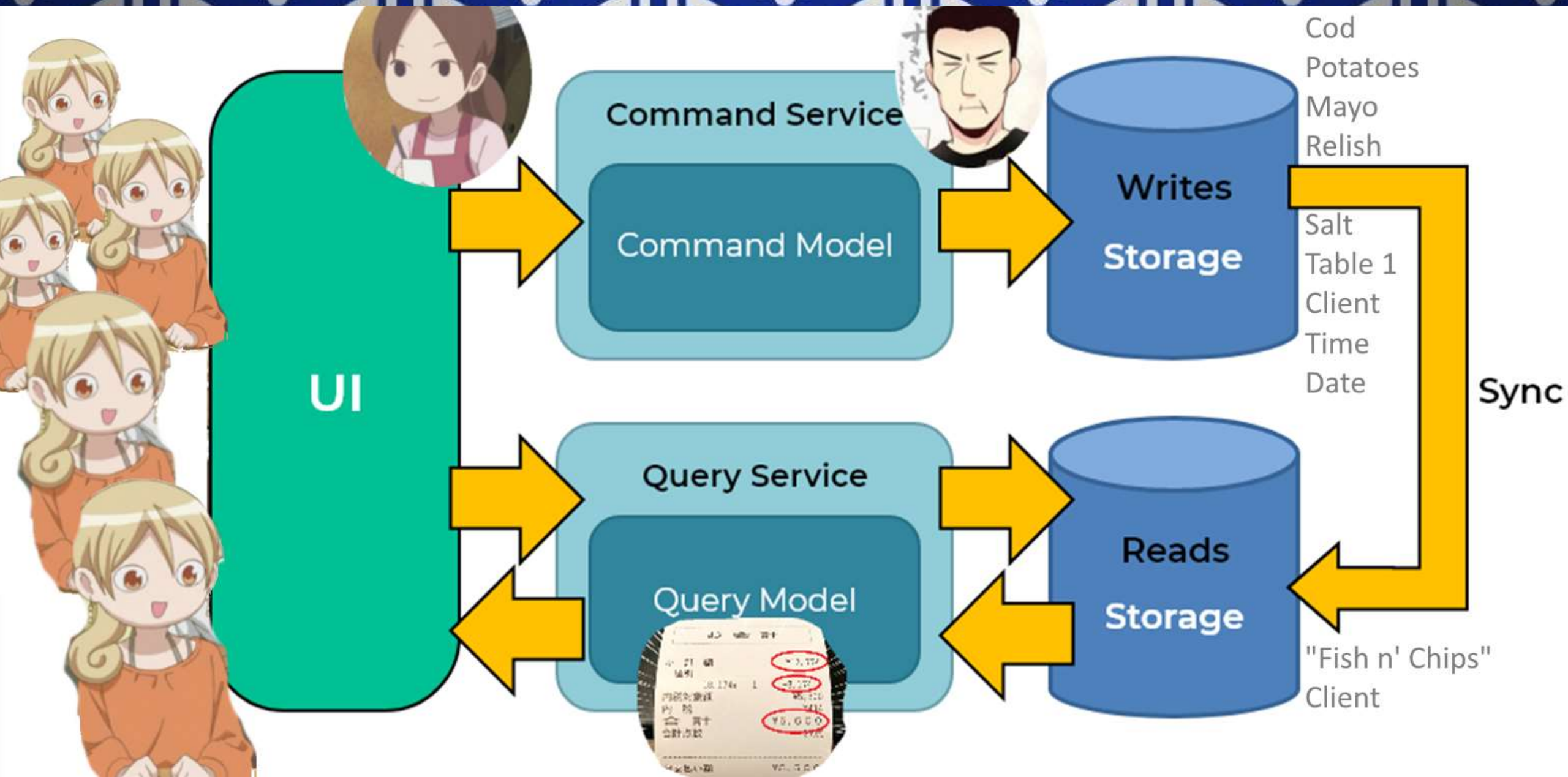
THE END

(See? You knew that story!)

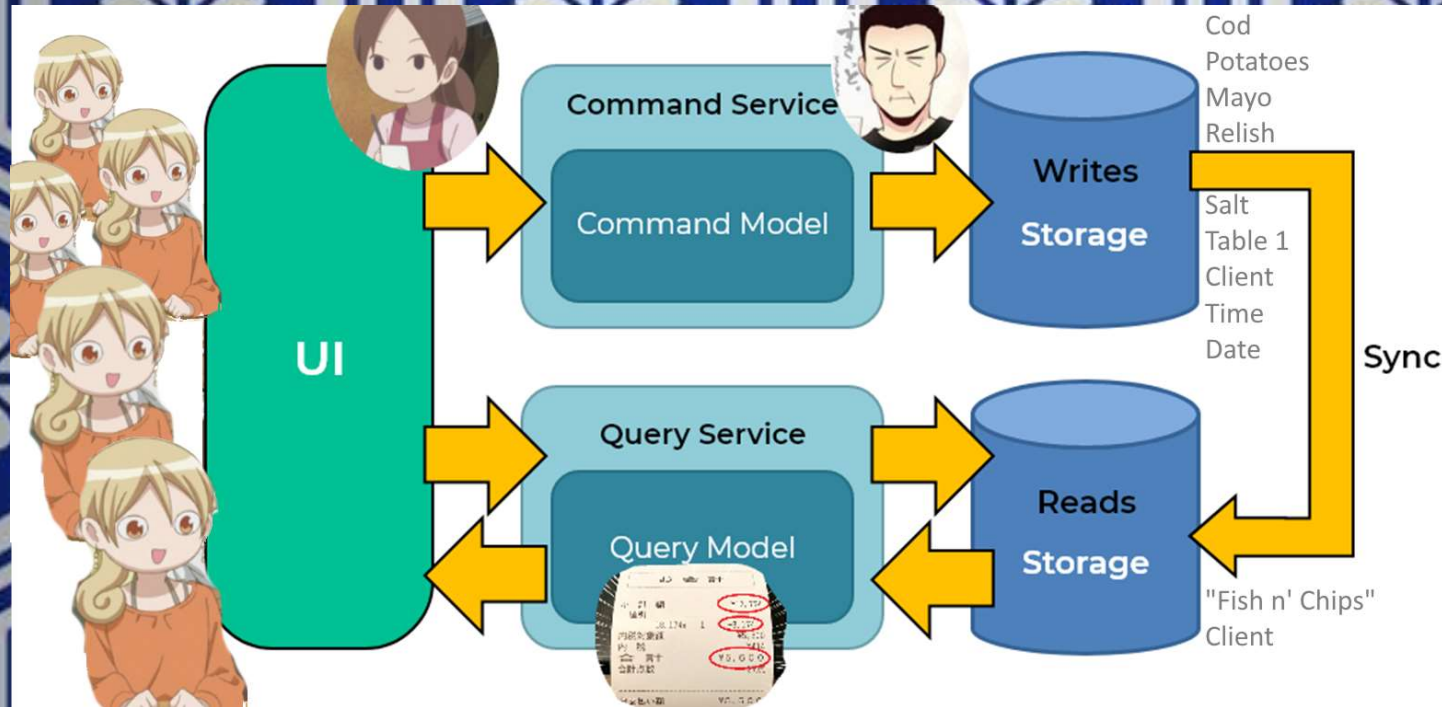


CORS

...is a similar story



See how many users?



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



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



QUESTION 1



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

AUTOFAC

(IOC)

(TOTOROMAGICALLYSUPPLYINGFUN)



AUTOFAC

The background of the slide is a dark, blue-toned illustration of the anime character Totoro. Totoro is depicted from the chest up, holding a large, dark umbrella over two small children. The children are looking up at Totoro. The scene is set against a dark, cloudy sky with some distant lights, suggesting a night or dusk setting.

- 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

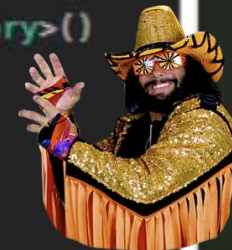
AUTOFAC

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 pet0 : base()
{
    _mediator = mediator;
    _logger = logger;
    _petQueriesRepo = petQueriesRepo;

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



PetStoreQueryController.cs

QUESTION 2



...how do they communicate internally?

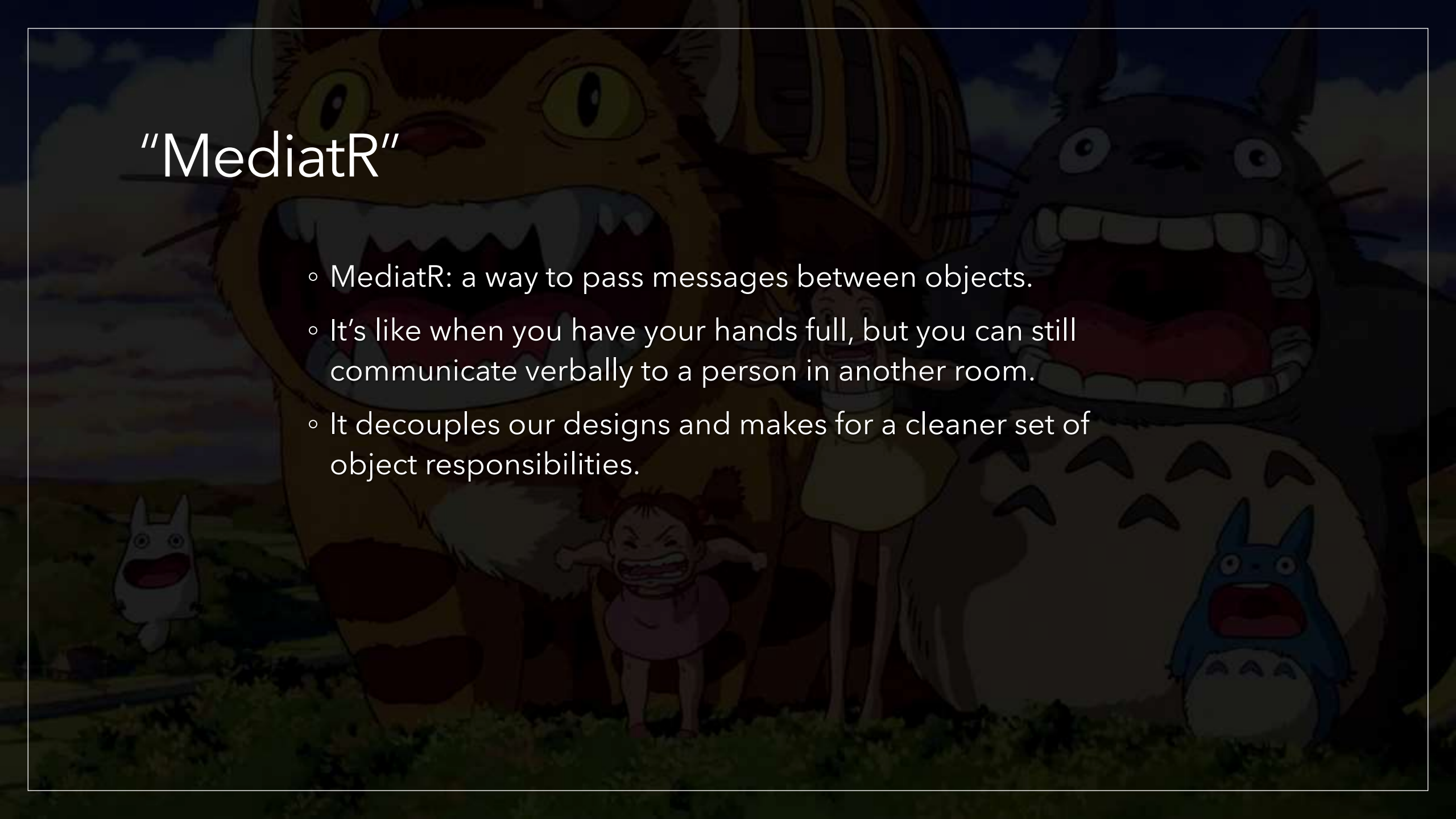
(Externally is another presentation, aka msg'ing between microservices)



"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"

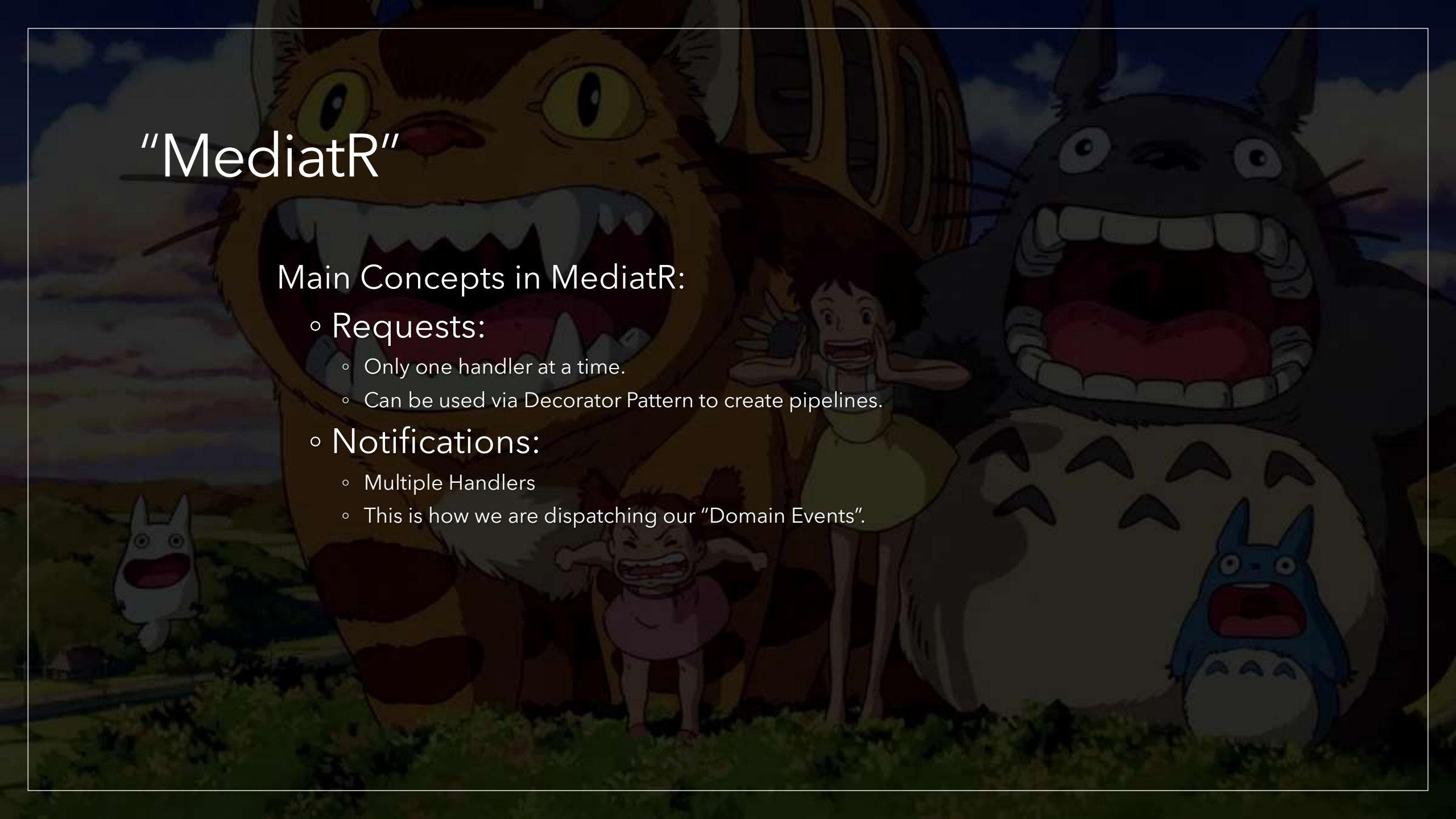
Main Concepts in MediatR:

- Requests:

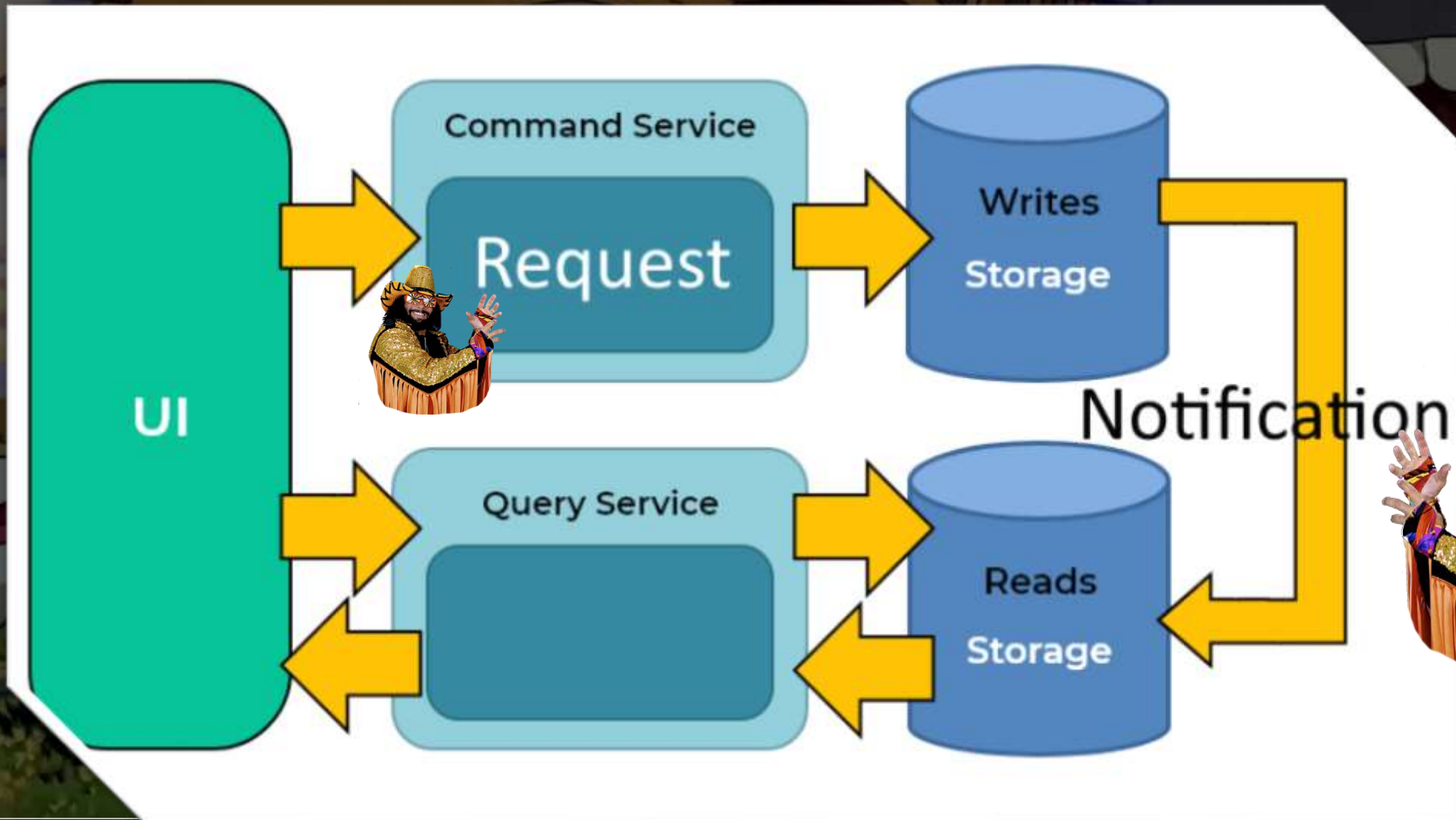
- Only one handler at a time.
- Can be used via Decorator Pattern to create pipelines.

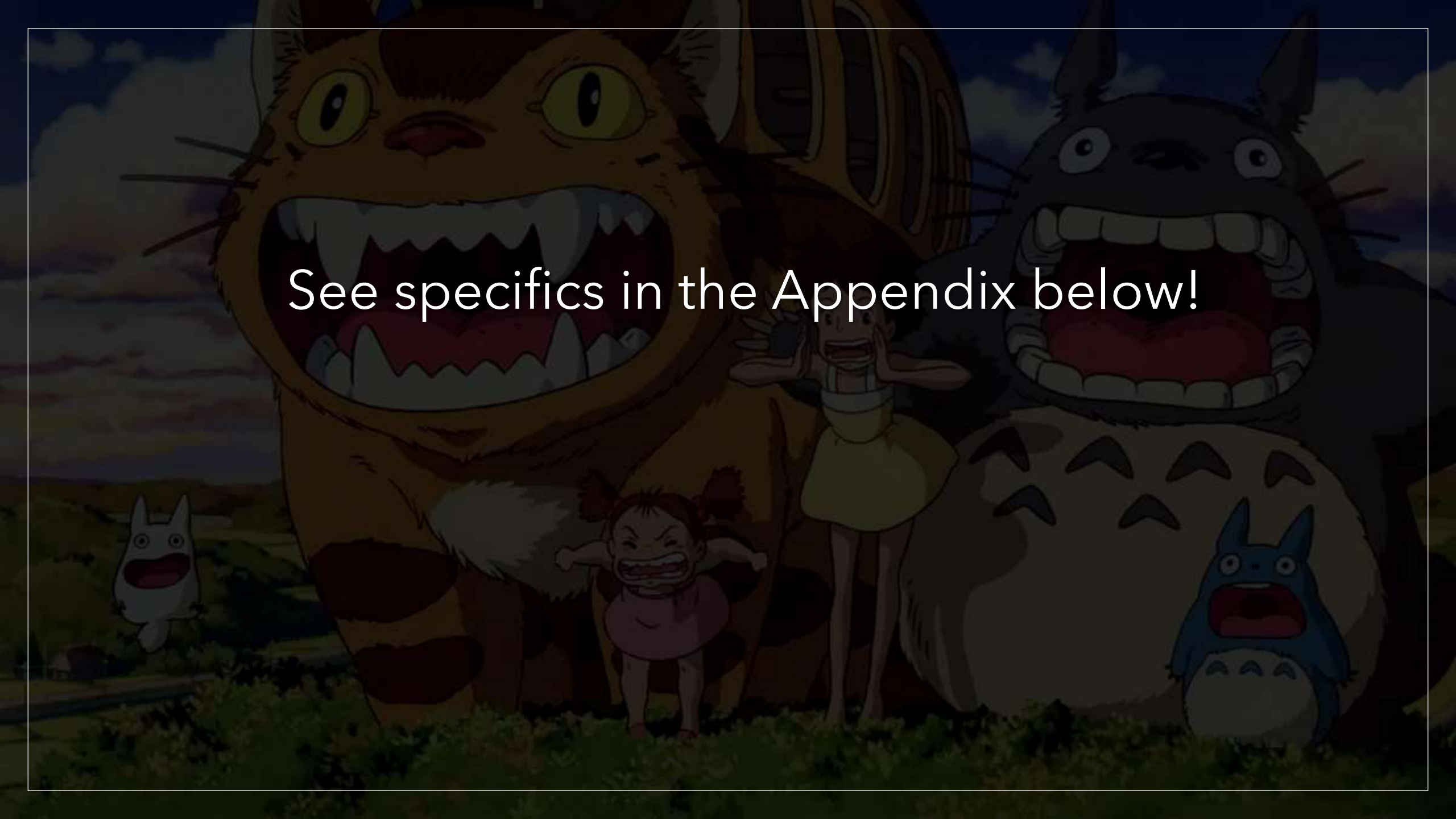
- Notifications:

- Multiple Handlers
- This is how we are dispatching our "Domain Events".



"MediatR"



A scene from Studio Ghibli's 'The Cat Returns' featuring a large orange cat, a girl in a yellow dress, and a girl in a pink dress, surrounded by Totoro-like characters.

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 guid
        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

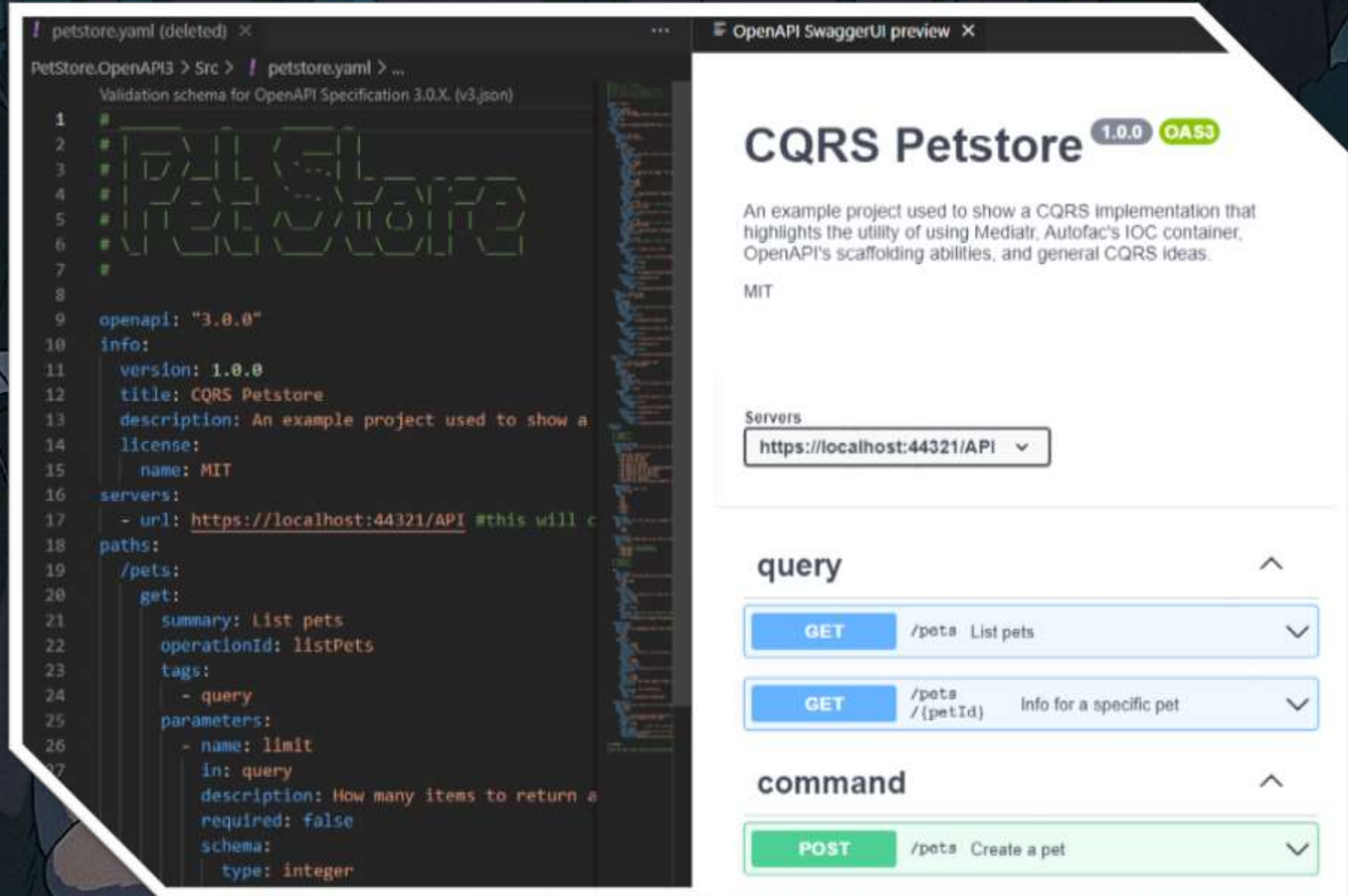
Editing / Testing

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](#)



The image shows a side-by-side comparison of an OpenAPI file and its visual representation. On the left, the VS Code editor displays a YAML file named `petstore.yaml` with the following content:

```
1 #
2 #
3 #
4 #
5 #
6 #
7 #
8
9 openapi: "3.0.0"
10 info:
11   version: 1.0.0
12   title: CQRS Petstore
13   description: An example project used to show a
14   license:
15     name: MIT
16 servers:
17   - url: https://localhost:44321/API #this will c
18 paths:
19   /pets:
20     get:
21       summary: List pets
22       operationId: listPets
23       tags:
24       - query
25       parameters:
26       - name: limit
27         in: query
28         description: How many items to return a
29         required: false
30         schema:
31           type: integer
```

On the right, the OpenAPI SwaggerUI preview shows the rendered interface for "CQRS Petstore 1.0.0" under the "OA53" standard. It includes a description: "An example project used to show a CQRS implementation that highlights the utility of using Mediatr, Autofac's IOC container, OpenAPI's scaffolding abilities, and general CQRS ideas." The license is MIT. A "Servers" dropdown is set to `https://localhost:44321/API`. Under the "query" section, there are two GET endpoints: `/pets` (List pets) and `/pets/{petId}` (Info for a specific pet). Under the "command" section, there is a POST endpoint: `/pets` (Create a pet).

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.

The screenshot displays the Nswag Studio application window. The title bar reads "petstore.nswag". The interface is divided into several sections:

- Input:** OpenAPI/Swagger Specification
- Runtime:** A dropdown menu is set to "NetCore31". Below it, a note states: "Specifies the used command line binary; should match the selected assembly type."
- Default Variables:** A text field contains "(foo-bar.baz-bar)", with a usage example: "usage: \$(foo)".
- Specification URL:** A text field contains "http://redocly.github.io/redoc/openapi.yaml", with a "Create local Copy" button to its right.
- Specification JSON/YAML (if specified, the URL is ignored):** A large text area displays the OpenAPI specification for the Petstore API. The JSON content is as follows:

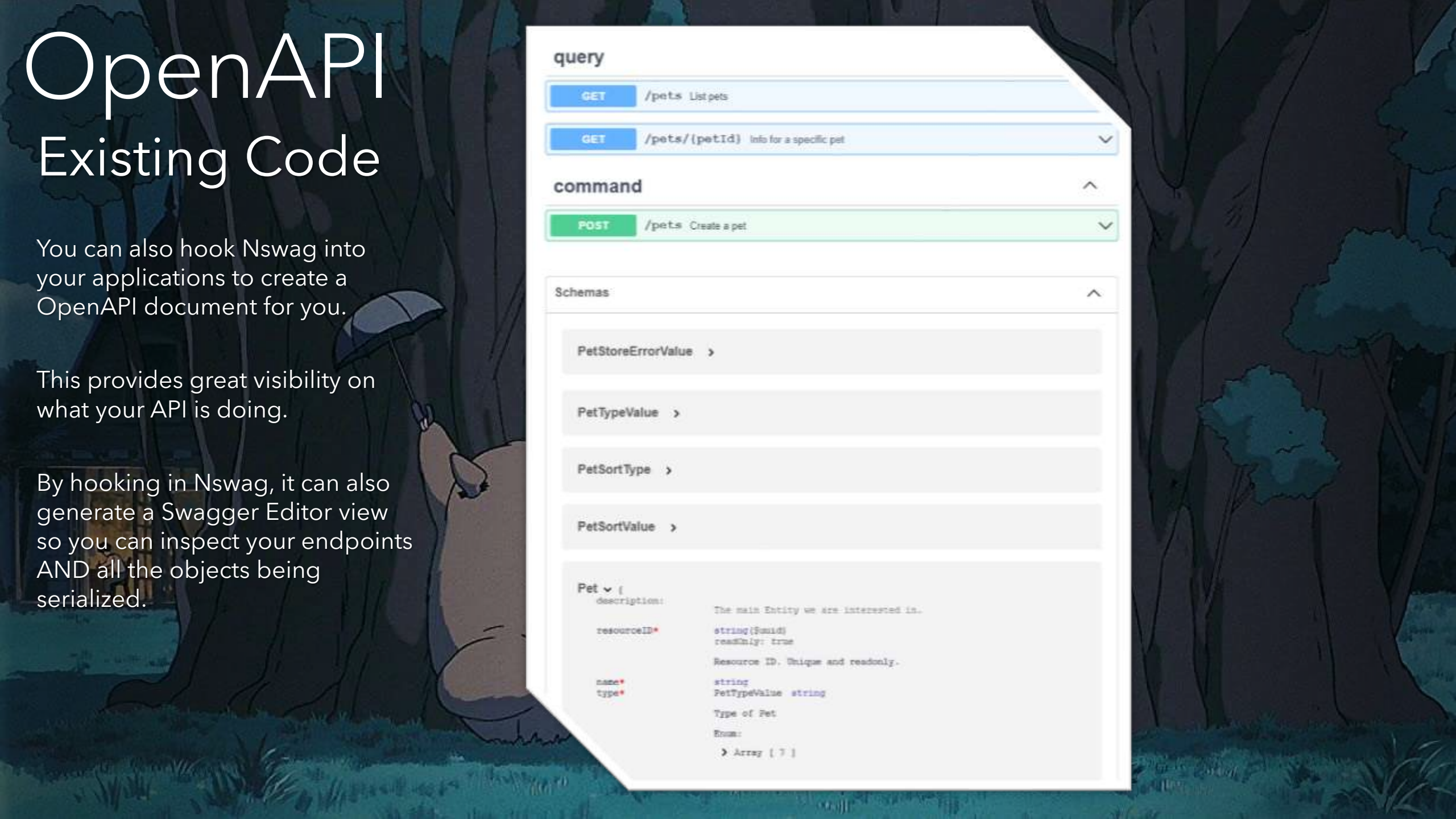
```
1 #
2 #
3 # PetStore
4 #
5 #
6 #
7 #
8
9 openapi: "3.0.0"
10 info:
11   version: 1.0.0
12   title: CQRS Petstore
13   description: An example project used to show a CQRS implementation that highlights the
14   license:
15     name: MIT
16 servers:
17   - url: https://localhost:44321/API #this will change.
18 paths:
19   /pets:
20     get:
21       summary: List pets
22       operationId: listPets
23       tags:
24         - query
25       parameters:
26         - name: limit
27           in: query
28           description: How many items to return at one time (max 100)
29           required: false
30           schema:
31             type: integer
32             format: int32
33         - name: offset
34           in: query
35           description: Defines the "page". The size of this page is indicated by the "lin
36           required: false
37           schema:
```
- Outputs:** A section on the right with checkboxes for "TypeScript Client", "CSharp Client" (checked), and "CSharp Controller" (checked). Below these are tabs for "OpenAPI/Swagger Specification", "CSharp Client", and "CSharp Controller".
- CSharp Controller Settings:** A panel on the right with various configuration options:
 - Namespace:** "Petstore.Common.Command"
 - Additional Namespace Usages (comma separated):** "System.Linq"
 - Controller Style:** "Abstract"
 - Controller Target:** "AspNetCore"
 - Use ASP.NET Core (2.1) ActionResult type as return type:** (checked)
 - Add cancellation token to controller methods:** (checked)
 - Add model validation attributes:** (checked)
 - Wrap success responses to allow full response access:** (unchecked)
 - Generate optional parameters (reorder parameters (required for** (unchecked)
 - Excluded Parameter Names (comma separated):** (empty field)
 - Route Naming Strategy:** "None"
 - Web API Controller:** (checked)
 - BasePath override (RoutePrefix):** (empty field)
 - Controller Class Name:** "{controller}"
 - Controller Base Class Name (optional, use 'Microsoft.AspNet** (empty field)
 - Operation Generation Mode:** "The {controller} placeholder of the Class Name is replaced by ger"

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.



query

- GET /pets List pets
- GET /pets/{petId} Info for a specific pet

command

- POST /pets Create a pet

Schemas

- PetStoreErrorValue >
- PetTypeValue >
- PetSortType >
- PetSortValue >

Pet {

- description: The main Entity we are interested in.
- resourceID* string(\$mid)
readOnly: true
Resource ID. Unique and readonly.
- name* string
- type* PetTypeValue string
Type of Pet
Enum:
Array [7]

DEMO

Let's see this baby this in action!

(Please clone my CQRS PetStore project.)



APPENDIX

Technology

Autofac: <https://autofac.org/>

Fluent Validation: <https://fluentvalidation.net/>

Martin Fowler: <https://martinfowler.com/bliki/CORS.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 ago | 1 author, 1 change
    public Pet Pet { get; }

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

•O 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.

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<,>));
```