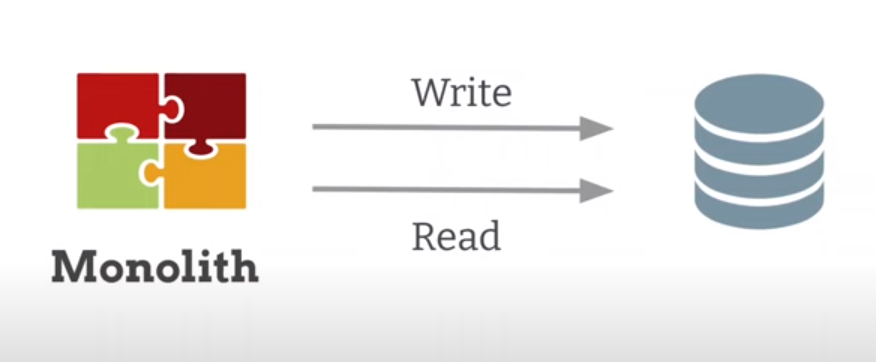
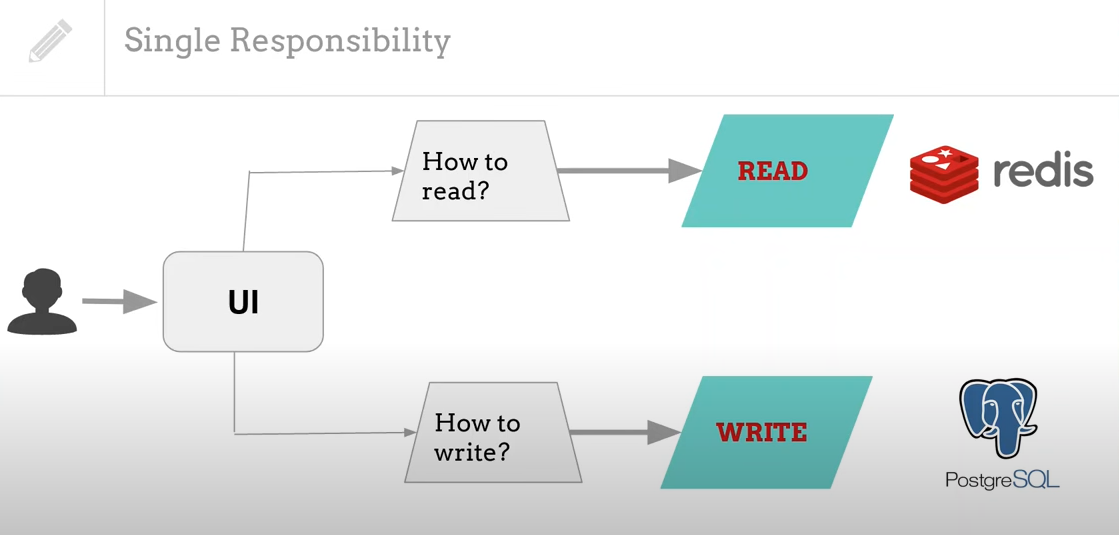
# CQRS (Command and Query Responsability Segregation)

Es el patrón de diseño más popular para arquitecturas de microservicios. Separa operaciones de lectura de operaciones de escritura a una base de datos.



En aplicaciones monolíticas una única base de datos responde a ambos tipos de operaciones, lectura y escritura de datos. Cuando el sistema crece, la complejidad de las consultas crece. Una consulta compleja podría unir muchas tablas y esto produciría un bloquero de unos segundos en esas tablas. Una operación de escritura impacta otras tablas produciendo otro bloqueo innecesario en una serie de tablas.

La idea del patrón CQRS, es segregar las responsabilidades de escritura y lectura. Se podría tener 2 bases de datos, una para leer datos y otra para escribir. Es una manera de aplicar el principio de responsabilidad única, en la arquitectura de una aplicación.



Por ejemplo, una base de datos puede ser relaciones y la otra una NoSQL

## COMMAND

Son acciones muy concretas, relacionadas con la creación o modificación de datos.

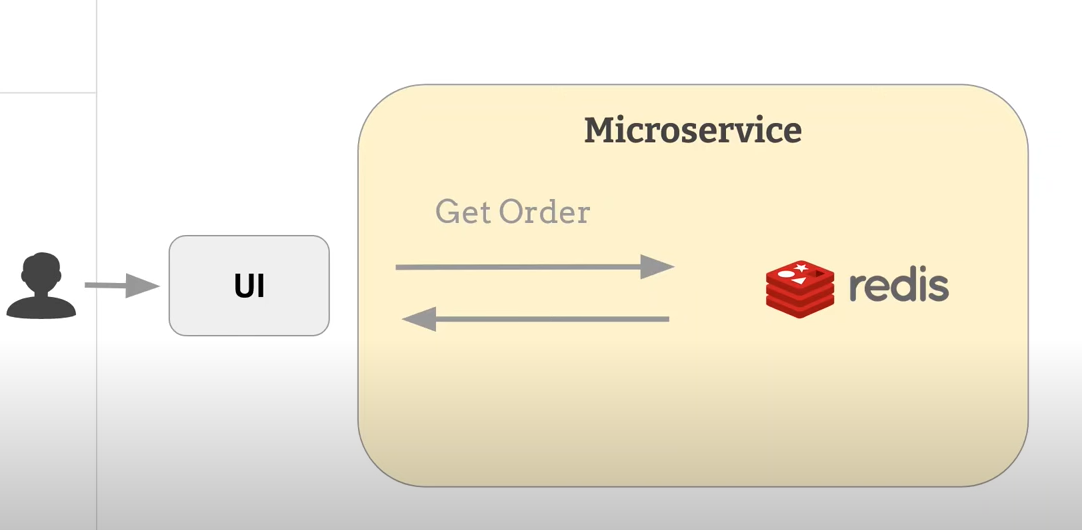
* “Update producto price”
* “Add producto to the cart”
* “Complete order”

## QUERIES (Nunca modifican la base de datos)

Siempre devolverán un json con el/los objetos DTO que vengan desde la base de datos

* “get order”
* “get producto price”
* “get products”

En CQRS ambos se mantienen independientes y aislados.



Si una orden es cancelada justo antes del delivery, es necesario involucrar otros procesos.

|  |  |
| --- | --- |
|  | * Cancelar la orden del depósito para que el envío no se haga. * Volver a poner el producto en stock * Si la orden ya fue pagada, se debe poner en marcha un proceso de reembolso del dinero. |

Para poder manejar esta lógica de negocio, se propone utilizar herramientas como, por ejemplo: “Command Handler” (Manejador de comandos) que determine qué acciones tomar cuando cierto comando se dispara.

## EVENT-DRIVEN ARCHITECTURE (Arquitectura Orientada a Eventos)

Cuando se produce una actualización en la base de datos, se dispara un evento que puede ser gestionado por un “message broker” (RabbitMQ/Kafka) y luego este evento será consumido por la base de datos de lectura que sincroniza sus datos con los últimos cambios.

Este mecanismo crea un problema de consistencia natural, porque el tipo de comunicación que implementa el “message broker” es asíncrono, por lo cual puede que la información no se refleje de manera inmediata en la base de datos de lectura. Esto se conoce como “Eventual Consistency” (Consistencia eventual).

## EVENT-SOURCING (Patrón de diseño)

La idea es almacenar en algún lugar todo el histórico de todos los eventos que escriben en la base de datos. Esto será tomado como la fuente de la verdad de los datos.

La base de datos de lectura, consumirá esos eventos para actualizar su propio estado.

*Sin Event-Sourcing*

|  |  |
| --- | --- |
|  | * Una aplicación normal guarda el último estado de los datos * El nuevo valor sobrescribe el anterior |

*Con “Event-Sourcing”* se guardan todos los eventos que modificaron los datos y se guardan en un orden secuencial, de tal manera que luego puedan ser reconstruidos por completo.

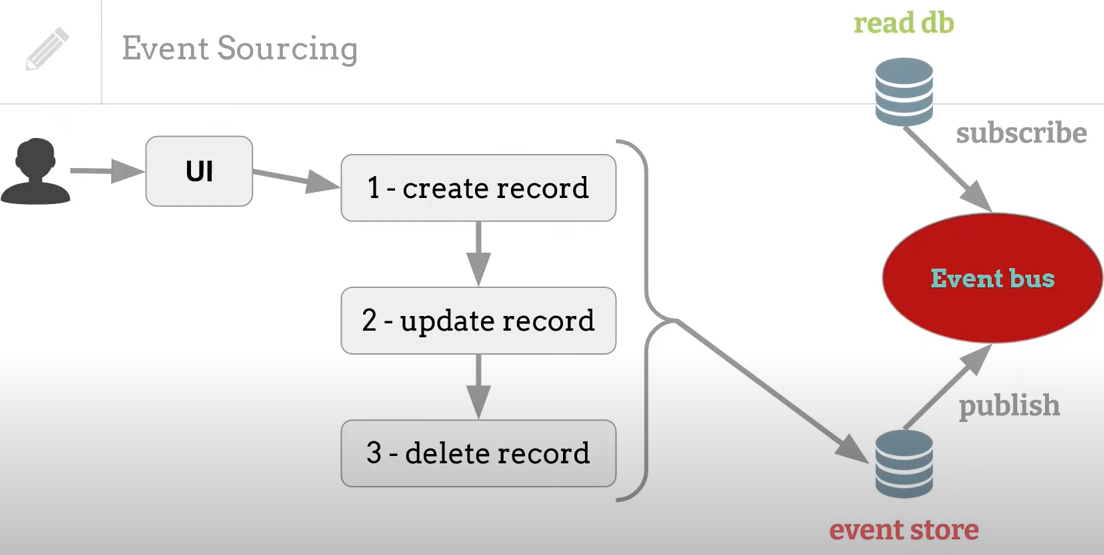
Esta base de datos de llama *“Event-Store”* en lugar de almacenar el estado de un dato, almacena cada cambio a una lista secuencial de eventos que modificaron el dato.

“Event-Store” se convierte en la fuente de la verdad de los datos. Luego todos los eventos se convertirán en datos en la base de datos de lectura.

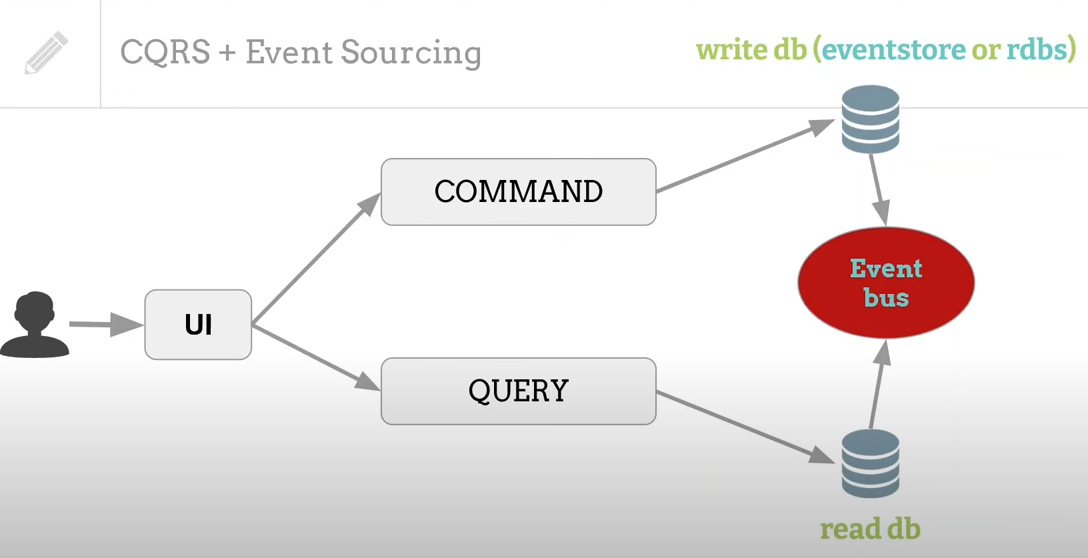
Esta operación de conversión, puede ser gestionada por algún patrón que pueda suscribir y publicar eventos al message broker.

|  |  |
| --- | --- |
| Beneficio | Desventaja |
| El ultimo estado de la base de datos, siempre se puede recrear en cualquier punto del tiempo, porque contiene el histórico de todos los cambios que alguna vez se realizaron | El tiempo que tomaría reconstruir una base de datos con una gran cantidad de eventos |

El usuario genera una serie de eventos, los cuales se almacenan en un “Event-Store”. Cada vez que un nuevo evento se almacena, el evento se publica en un “Broker” o un “Pass” y desde allí la base de datos de lectura tendrá un suscriptor que notificará que un nuevo evento se ha guardado y podrá actualizar el estado de sus datos.



## PATRON CQRS + PATRON EVENT-SOURCING PARA MICROSERVICIOS



El “EventBus” se encargará que los datos en la base de datos de lectura estén siempre sincronizados con los de escritura.

# PRACTICA

1. Crear una aplicación ASP NET Core Web Api
2. Instalar los nugget:

* Microsoft.EntityFrameworkCore.InMemory (Permite trabajar con una base de datos en memoria)
* MediatR 11.1.0 (Permite implementar el patrón mediador. Va a mediar entre los requests que provienen del api y la lógica de negocio y lo va a hacer utilizando handlers)
* MediatR.Extensions.Microsoft.DependencyInjection 11.1.0
* Autofac.Extensions.DependencyInjection

1. Crear 3 folders

* Application
  + DTOs: Va a permitir crear un modelo o una clase específica que será la que va a viajar entre capas y no representa el dominio exacto.
  + Handlers: Donde va a estar la lógica de negocio.
* Domain: Donde va a estar el modelo y las entidades del dominio.
* Infraestructure:
  + Commands: Acá estarán los comandos
  + Queries: Acá estarán las consultas.

1. Modificar la clase ‘Program.cs’ así:

//TODO: Mediator recibe como parámetro del assembly en el cual va a encontrar todos

//los comandos, queries y handlers y todos los objetos que va a estar mediando.

//TODO: Conecta los comandos con su handler, las queries con su handler.

builder.Services.AddMediatR(typeof(Program).Assembly);

//TODO:Se registra una base de datos en memoria con el nombre de 'TaskDb'

builder.Services.AddDbContext<ApplicationDbContext>(options=>

options.UseInMemoryDatabase("TaskDb"));

Todo antes de:

var app = builder.Build();

1. Modificar la clase ‘CreateTaskCommand.cs’ así:

public record CreateTaskCommand(string Title, string Description) : IRequest<TaskItemDto>;

'TaskItemDto' va a ser el tipo de respuesta que va a tener este comando y así va a ser para todas las queries y comandos.

'Title' y 'Description' son los elementos que se quiere que tenga el comando (que reciba o espere el comando).

‘record’ fue agregado en C# 6 y tiene la ventaja de que son inmutables.

Una vez creados no pueden cambiar.

1. ‘UpdateTaskCommand.cs’:

public record UpdateTaskCommand(int Id, string Title, string Description, bool IsCompleted) : IRequest<TaskItemDto>;

1. ‘DeleteTaskCommand.cs:

public record DeleteTaskCommand(int Id) : IRequest<bool>;

1. ‘GetAllTaskQuery.cs’

public record GetAllTaskQuery : IRequest<IEnumerable<TaskItemDto>>;

1. ‘GetTaskByIdQuery.cs’:

public record GetTaskByIdQuery(int Id) : IRequest<TaskItemDto>;

1. Crear los handlers. Uno por cada acción (GetAll, GetById, Create, Update, Delete)
2. Crear el controlador e implementar cada método, utilizando los comandos para cada una de las acciones.

Se debe inyectar Mediator en cada controlador.

En la implementación de cada ActionResult y al método ‘Send’ se le dice cuál es el comando o la query que se debe utilizar.