Tarea 3

CENFOTEC

Johnny Rivera

Desarrollo de RESTful Web APIs con .NET Core

Trabajo de investigación

1. Investigar sobre el significado y concepto de CQRS en ASP.NET Core
2. Investigar sobre el uso de la libreria FluentValidation para ASP.NET Core y aplicarlo en el codigo indicado por el profesor cuando sea indicado.
3. Subir un TXT con la respuesta al item 1 de la tarea y el ejercicio resuelto a la carpeta /entregables/tarea-03 correspondiente en su repositorio de estudiante

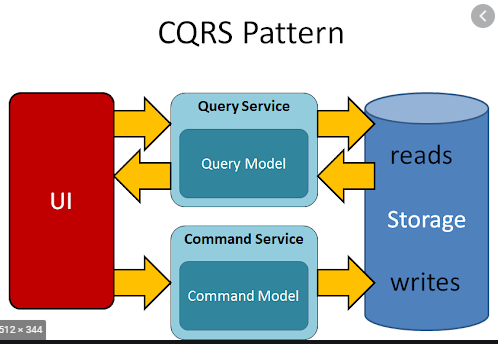
Respuesta 1 investigado de la fuentes:

<https://es.slideshare.net/julitogtu/introduccin-a-cqrs>

<https://eamodeorubio.wordpress.com/2012/09/03/cqrs-1-que-es/>

**CQRS** En una arquitectura tradicional, tenemos un único sistema que se encarga de realizar operaciones de negocio y nos permite consultar la información en la que se encuentra nuestro sistema. Sin embargo, esto viola el **principio de única responsabilidad (SRP)**, ya que el mismo sistema se encarga de hacer dos cosas que en principio son muy distintas: exponer operaciones que evolucionen el estado del sistema de forma consistente, y leer el estado del sistema. Estas dos responsabilidades tienen requisitos muy diferentes desde distintos puntos de vista: funcional, escalabilidad, tiempo de respuesta, seguridad, criticidad, etc. Si nos tomamos en serio el principio de separación de responsabilidades, y somos partidarios de arquitecturas modulares en vez de sistemas monolíticos y centralizados, debemos buscar  una arquitectura alternativa.

**Command Query Responsability Segregation (CQRS)**, es un estilo arquitectónico en el que tenemos dos subsistemas diferenciados, uno responsable de los comandos, y otro responsable de las consultas. Por comando entendemos una petición por parte del usuario u otro sistema, para realizar una operación de negocio, que evolucione el sistema de un estado a otro. Cada uno de estos subsistemas tiene un diseño, modelo de información y mecanismo de persistencia diferente, optimizado para las tareas que deba afrontar. Normalmente el subsistema de consulta suele ser mucho más simple que el otro. Veamos la siguiente figura para explicar un poco como funciona esto



El subsistema de comandos, simplemente recibe peticiones de comandos, valida que éstos son consistentes con el estado actual del sistema, y si es así los ejecuta. Como resultado de la ejecución de un comando, el estado del sistema cambia, y ese cambio se comunica al subsistema de consultas mediante algún mecanismo de sincronización.

El subsistema de consulta recibe los cambios en el estado del sistema mediante el mecanismo de sincronización. Durante la etapa de filtrado ignora los cambios en los que no está interesado. En algunos mecanismos de sincronización esta etapa puede formar parte de la configuración del mismo, y no del código de aplicación. Después los cambios se pueden pasar por una etapa opcional donde se pueden transformar, añadirle información calculada y agregar información varios cambios. De nuevo en algunos casos el mismo mecanismo de sincronización podría proporcionarnos herramientas para definir esta etapa de forma declarativa. Finalmente se actualiza de forma adecuada la base de datos. La ejecución de la consulta simplemente consiste en exportar los datos de la BBDD como DTOs y serializarlos en un formato adecuado.

Si en el subsistema de consulta usamos una BBDD relacional, podríamos definir un esquema optimizado para las consultas, de forma que estas no necesiten joins, y las sentencias SELECT sean muy simples y sencillas de optimizar. Tal vez una tabla por consulta, con una columna por parámetro de consulta, y una columna adicional con el resultado en formato ya directamente serializado. Evidentemente otras opciones son posibles.

Y ya está, cualquier sistema que cumpla lo descrito se puede considerar CQRS. Ahora bien, como el demonio está en los detalles, existen muchos puntos importantes a decidir:

* La naturaleza exacta del mecanismo de sincronización entre el subsistema de comandos y el de consultas. Se puede usar algo tan simple como una llamada local intraproceso, o algo más sofisticado como un sistema de colas, o un servicio web (REST o SOAP). También hay que decidir si la sincronización será pull o push, es decir, ¿el subsistema de comandos notificará al de consulta los cambios? ¿O tal vez el subsistema de consultas preguntará de forma periódica que cambios se produjeron desde la última vez?
* El paradigma de persistencia de cada subsistema. Puede ser SQL tradicional, o noSQL. Lo interesante es que cada subsistema puede usar el paradigma que más le interese. En concreto si usamos relacional para las consultas, se nos plantea la disyuntiva de usar un esquema desnormalizado totalmente optimizado para las consultas (una tabla por consulta con las columnas específicas a recuperar por ésta) o bien un esquema en tercera forma normal que nos de más flexibilidad.
* Si el subsistema de comandos tiene persistencia o no. En realidad el subsistema de comandos sólo necesita consultar el estado para saber si puede ejecutar un comando o no. Esta consulta podría proceder del subsistema de consulta, en vez de una BBDD propia. Esto puede ser una opción según que modelo de consistencia sea admisible.
* Transaccionalidad en el subsistema de comandos. ¿Cuándo se da por terminada la transacción representada por el comando? ¿Cuándo el nuevo estado del sistema se ha modificado en memoria? ¿O tal vez, cuando este estado se ha persistido en la BBDD específica del subsistema de comandos? ¿Quizás sería mejor esperar a que el subsistema de consulta haya recibido el cambio y actualizado el modelo de consulta? Esta decisión es crítica a la hora de definir el modelo de consistencia de datos que queremos (ACID o BASE).
* ¿Un sólo subsistema de consulta o varios? La arquitectura CQRS nos permite definir varios subsistemas de consultas, especializados en cosas diferentes. Cada uno puede tener un enfoque diferente en cuanto a la persistencia y consistencia. Por ejemplo uno podría usar un sistema noSQL para consultas específicas, y otro podría ser un sistema OLAP para hacer minería de datos.
* ¿Un sólo subsistema de comandos o varios? De nuevo CQRS nos da la libertad de tener distintos subsistemas especializados en distintas familias de comandos. Uno para procesar comandos relacionados con los pedidos, otros para el control del stock, otro para pagos.

Muchas decisiones, muchas combinaciones, cada una de ellas con su correspondientes ventajas e inconvenientes. Los siguientes posts de la serie tratarán de explorar un poco estos aspectos.

El usar CQRS nos da los siguientes beneficios:

* Ambos subsistemas pueden evolucionar por separado; el mantenimiento y despliegue puede estar diferenciado. Esto es una ventaja porque normalmente el ritmo de cambios en la funcionalidad es muy diferente en las consultas y los comandos. En mi experiencia es más frecuente añadir una nueva consulta que un nuevo comando.
* Ambos sistemas pueden escalar por separado. Si nuestro negocio es muy intensivo en lecturas, podemos dedicar más máquinas al subsistema de consultas de forma sencilla. Simplemente levanta otra instancia y conéctala de forma que pueda recibir los cambios transmitidos por el subsistema de comandos.
* El estilo CQRS es modular de forma inherente. Podemos escoger ir a una arquitectura modular, donde haya varios subsistemas de consulta y varios subsistemas de comandos. Cada uno de ellos de nuevo puede ser diseñado, mantenido, escalado y desplegado por separado. Es muy interesante la posibilidad de añadir nueva funcionalidad (nuevos subsistemas de comandos o consultas) sin tener que parar el resto del sistema.

Sin embargo todo esto no es gratis:

* Es más complejo, sobre todo debido a que debemos diseñar con cuidado el mecanismo de sincronización en los comandos y las consultas.
* El tener sistemas de persistencia separados para los comandos y las consultas, o el tener múltiples subsistemas de cada uno, puede provocar que tengamos cierta duplicación de información. Esto puede ser negativos si estamos preocupados por ahorrar espacio en disco.
* Para construir el subsistema de comandos necesitamos saber que comandos existen, como evolucionan el estado del sistema y cuando la ejecución de un comando es consistente con el estado del sistema. Todo esto exige modelar bien el negocio, olvidarse de las tablas y centrarse bien en las acciones del usuario y modelar bien los casos de uso como máquinas de estado o similar. O sea, no sirve para hacer aplicaciones de mantenimientos de tablas o CRUD. El problema es que muchos frameworks de moda (ROO, GRAILS, RAILS…) están optimizados para ser productivos haciendo CRUD, no para CQRS. No digo que sea imposible usar, por ejemplo RAILS, pero sí es verdad que no va a ser tan productivo.

Los dos últimos puntos son dificultades relativas. El tener duplicación de información y distintos subsistemas es bueno desde el punto de vista de la redundancia y la disponibilidad de la información. Si un subsistema de consultas se corrompe, se puede reconstruir a partir del subsistema de comandos. Por otro lado el no poder usar un enfoque CRUD, o que los frameworks de moda no nos faciliten tanto la tarea, lo veo más un problema social que de ingeniería

**Uso de la librería FluentValidation para ASP.NET Core**

Usaremos la librería Fluent Validation para validar los parámetros que ingresan a nuestra API.

Ejemplo práctico:

project.json add:

"FluentValidation.AspNetCore": "6.4.0-beta3"

startup.cs

services

.AddMvc()

.AddFluentValidation(fv => fv.RegisterValidatorsFromAssemblyContaining<Startup>());

Validation:

public class Foo

{

public string Bar {get; set;}

}

public class FooValidator : AbstractValidator<Foo>

{

public FooValidator()

{

RuleFor(x => x.Bar).NotEmpty().WithMessage("Error Message");

}

}

[**FluentValidation**](https://fluentvalidation.codeplex.com/)

FluentValidation es una librería en donde por medio de Expresiones Lambda podemos fácilmente construir reglas de validación poderosísimas, con mucha facilidad y con una flexibilidad enorme, FluentValidation puede servirnos tanto para proyecto del tipo WindowsForms o como para aplicaciones Web usando ASP.NET o ASP con MVC, por lo cual si usamos una arquitectura 3 capas en un proyecto WindowsForms y el día de mañana deseamos migrar nuestra capa de presentación a un proyecto web, no tendremos que preocuparnos por cambiar la manera de validar nuestras reglas de Datos debido a la compatibilidad entre estos.