CQRS

Segregación de Consultas y Comandos

Oriol Ylla-Català Rosell y

Adrià Velardos Palomar

Desarrollo Eficiente del Software

Febrero 2018

Qué es?

CQRS es un patrón de diseño a alto nivel que define una arquitectura que permite desarrollar código mantenible, de alto rendimiento y fomenta el desacople de las distintas capas de un proyecto.

En la mayoría de arquitecturas tradicionales el sistema se encarga tanto de hacer consultas al sistema de base de datos como de realizar acciones de negocio, o escribir en la base de datos. Esto infringe el principio de responsabilidad única (SRP en inglés), debido a que el sistema realiza dos tareas que son muy distintas entre ellas y que tienen requisitos particulares.

Así pues, la segregación de consultas y comandos (CQRS en inglés) nace con la finalidad de solucionar el acople de las dos funcionalidades dividiendo el sistema en dos subsistemas diferenciados para cada acción. En la figura 1 se puede observar esta división en subsistemas:

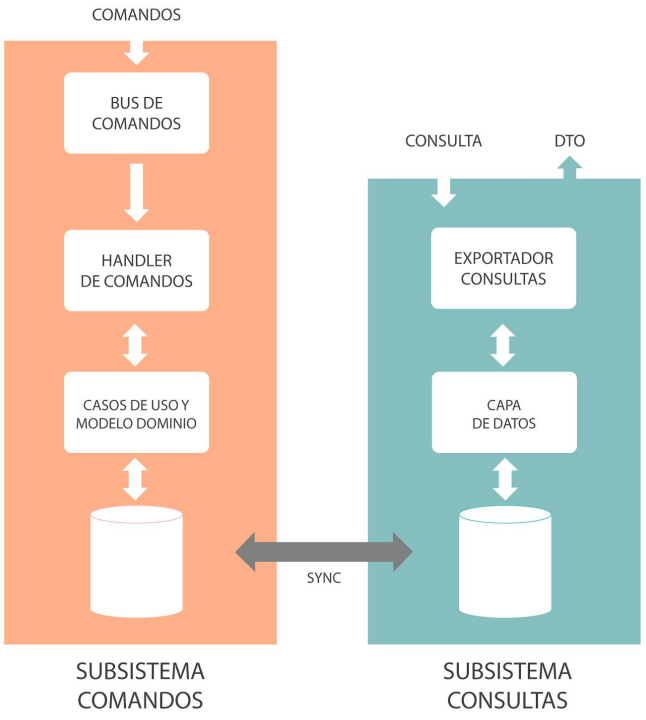


Figura 1

El primer subsistema se hará cargo de los comandos, es decir, realizar operaciones de negocio y inserciones en la base de datos. El segundo se encargará únicamente de consultar a la base de datos. Veámos cada subsistema en detalle.

Subsistema Comando

Este subsistema recibe comandos, los valida y si son consistentes con el estado actual del sistema los ejecuta. Como resultado de ejecutarlos el estado del sistema puede haber cambiado. Por ejemplo, un cambio de estado podría ser persistir nuevos datos en la base de datos. Este cambio se propaga mediante algún mecanismo de comunicación al subsistema de consultas para actualizar la base de datos. En el ejemplo de la figura 2 la flecha gris podría conectar con un bus de eventos y su *handler* asociado para que modificara los nuevos datos en la base de datos del subsistema de consultas.

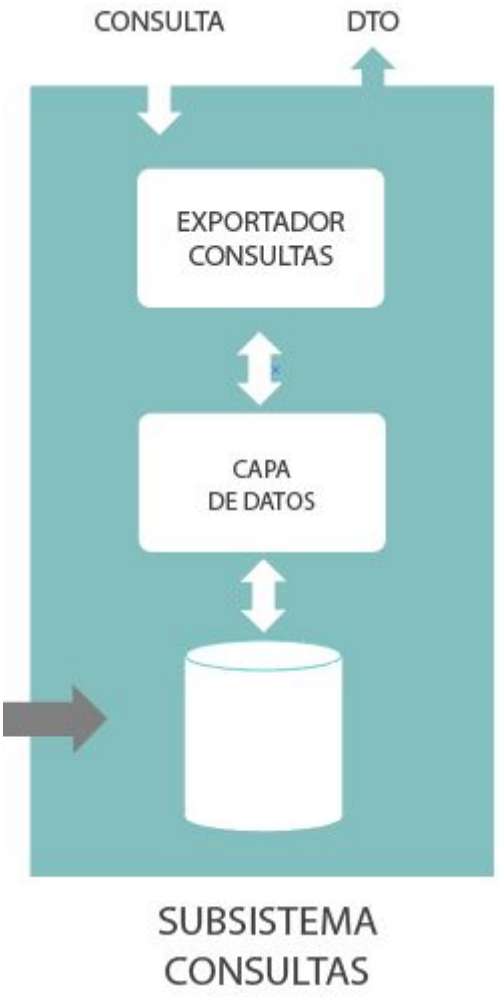
Figura 2

La ventaja que nos proporciona el hecho de tener a parte el subsistema comando del de consulta es que los objetos de dominio no tienen porqué exponer su estado interno al ser consultados.

Los comandos son instrucciones dirigidas a una entidad para que esta realice una acción. Suponiendo, como se puede observar en la figura 2, que estos comando vienen dados desde la interficie de usuario, pasarían por un bus de comandos, a la espera de que un *handler* los ejecute en el dominio apropiado. Los *handlers* o manejadores deben de poder conectar con el repositorio para cargar correctamente la entidad idónea para cada comando.

Subsistema Consultas

El subsistema de consultas representa la capa de lectura; es el encargado de hacer las peticiones a la base de datos, sin cambiar el estado del sistema. Contiene los métodos que devuelven DTO's que representan los datos que suelen ser presentados en la pantalla del cliente. Normalmente estos DTO's son proyecciones de objetos del modelo de dominio, lo que añade complejidad al sistema si estos objetos son complejos.



Gracias a tener la lógica de lectura separada en una capa, podemos saltarnos el modelo del dominio y obtener los datos directamente de la capa de lectura con una simple llamada que devuelve un DTO.

Normalmente las peticiones de datos son mucho más frecuentes que las ejecuciones del comportamiento del dominio, por lo que tener estas peticiones separadas en un modelo propio nos da mayor eficiencia y mantenimiento de las consultas.

Eventos de comunicación interna

Cuando el cliente ejecuta un comando, este pasa por un *handler* que carga la entidad que realiza la acción. Esta acción levanta un evento que subscribirà en un bus de eventos, y será gestionado por *handlers* específicos suscritos al bus. Este tipo de eventos son llamados eventos internos, y representan acciones en el estado que han ocurrido en el

pasado. El *handler* que realiza la acción no debe hacer ninguna lógica a parte de escrivir el estado.

Los eventos internos sirven para persistir el estado de una entidad guardando el flujo de eventos, lo que nos permite reconstruir estados pasados. Éstos no pueden modificarse o borrarse, ya que son de escritura única, por lo que si algún error en el comportamiento del dominio genera eventos erróneos, se deberá crear un evento que rectifique los errores. A éste patrón se le llama *Event Sourcing*.

No obstante, dependiendo de los requerimientos del sistema, este patrón puede representar demasiada complejidad a cambio de una ventaja poco importante, ya que se podría encontrar alternativas para saber estados anteriores, por ejemplo mediante atributos específicos en la entidad del dominio o escribiendo en un fichero de log.

Venatjas

- Escalado independiente de cada subsistema: al no depender un subsistema del otro se puede escalar cada uno de ellos de forma independiente según la carga de trabajo de lectura o escritura.
- DTO optimizados: el subsistema de consultas puede usar un esquema de datos que esté optimizado para su fin y del mismo modo, el subsistema de comandos puede estar optimizado para hacer actualizaciones.
- Seguridad: resulta más sencillo asegurar que solo las entidades de dominio correctas realizan persistencias de datos.
- Separación de funcionalidades: el hecho de separar la lectura de la escritura facilita que los modelos sean flexibles. Como la lógica de negocio compleja se implementa en el subsistema de comandos, el subsistema de consultas puede ser mucho más sencillo.
- El testeo se convierte en un proceso mucho más sencillo, incluso en las partes más complejas de la lógica de negocio.

Desventajas

- Complejidad: aunque la idea de CQRS es sencilla, el diseño del sistema puede resultar más complejo. Sobretodo si se añade el control de eventos, como en el

ejemplo de este documento. Ya que el sistema deberá de gestionar errores de mensaje o casos de duplicidad en la sincronización de ambas bases de datos.

- Coherencia: al separar la base de datos del sistema en dos, se podría dar el caso que los datos del subsistema de consultas estén obsoletos.

Cuando usar CQRS

La arquitectura CQRS resulta de utilidad para los dominios de colaboración en los que varios usuarios realizan lecturas y escrituras sobre los mismos datos de forma asimétrica.

También resulta útil en escenarios en los que un equipo de desarrolladores puede focalizarse en un modelo de dominio complejo que forma parte del subsistema de comandos, y otro equipo puede centrarse en el subsistema de lectura y en la representación de datos.

Para finalizar, CQRS no tiene porqué aplicarse en todo el sistema de nuestro proyecto, solo en los casos donde se obtenga un valor evidente de la separación de lectura y escritura de la base de datos. En caso contrario, se estaría añadiendo complejidad al sistema.

Fuentes

https://docs.microsoft.com/es-es/azure/architecture/patterns/cqrs
https://docs.microsoft.com/es-es/azure/architecture/guide/architecture-styles/cqrs
https://msdn.microsoft.com/es-es/magazine/mt147237.aspx
https://www.codeproject.com/Articles/555855/Introduction-to-CQRS
https://es.wikipedia.org/wiki/Command%E2%80%93query_separation
https://eamodeorubio.wordpress.com/2012/09/03/cqrs-1-que-es/
http://cqrs.nu/Faq