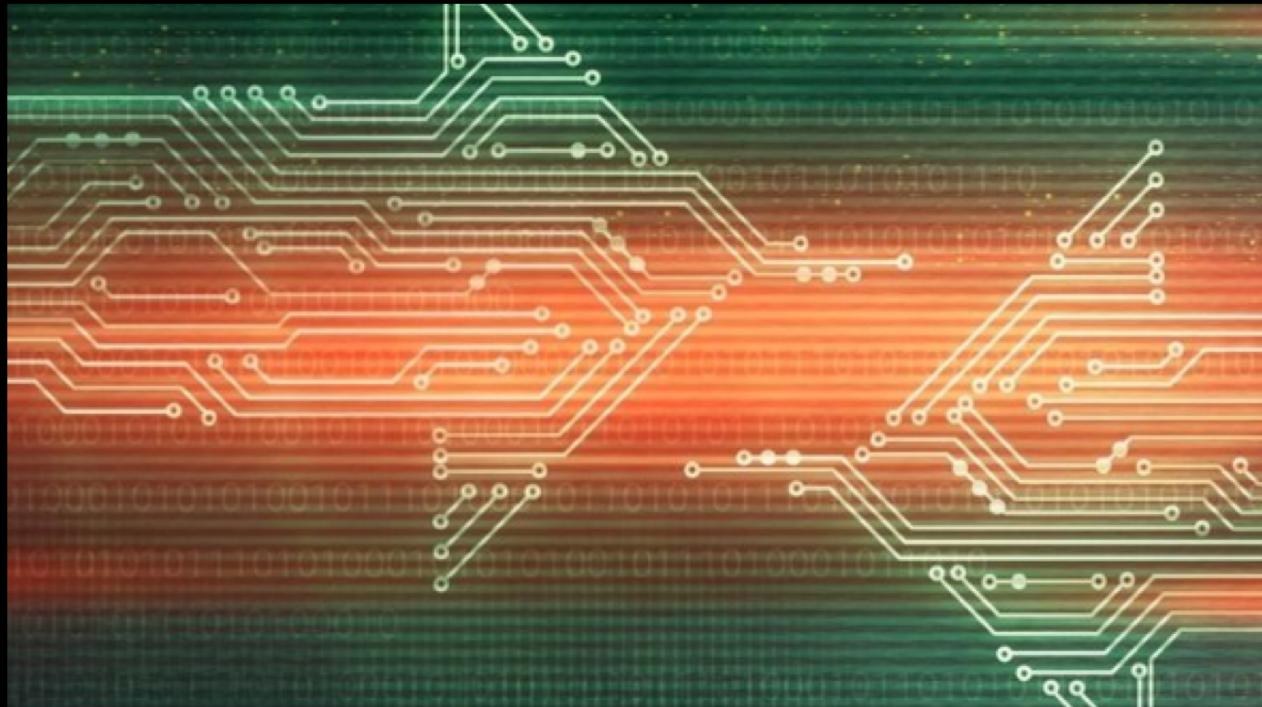


UN VISTAZO A LOS PATRONES CQRS Y EVENT SOURCING



VIERNES, 24/01/2020 CEEIM

El problema inicial



Usuarios pueden cambiar los procesos de negocio sin cambiar la aplicación

Operaciones CRUD básicas



- Usuarios deben conocer todos los detalles de los procesos de negocio.
- Nuestro dominio se limita a insertar, actualizar o eliminar datos
- Problemas de rendimiento

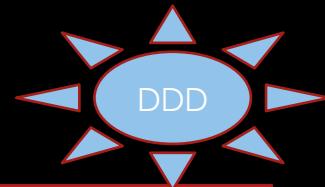
Data Centric Application

Orientadas a
CRUD

CQS

DDD

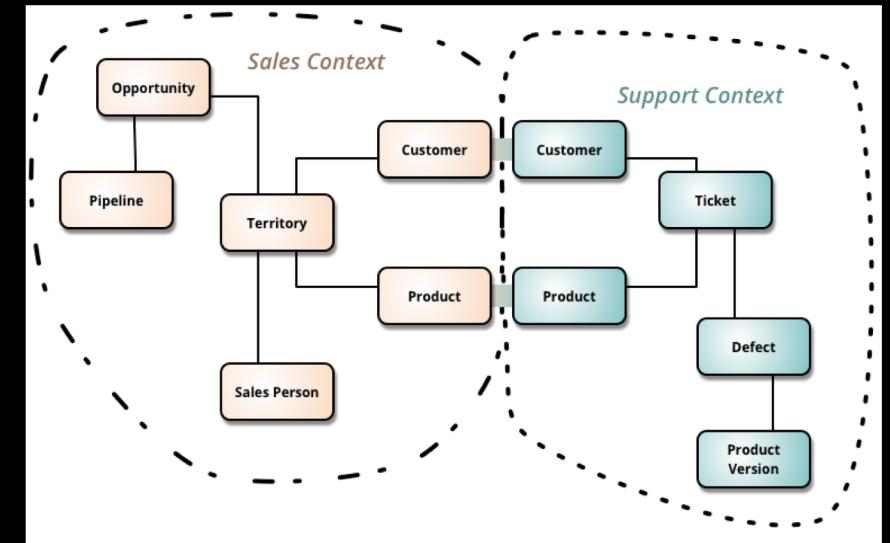
Orientadas a
comportamiento



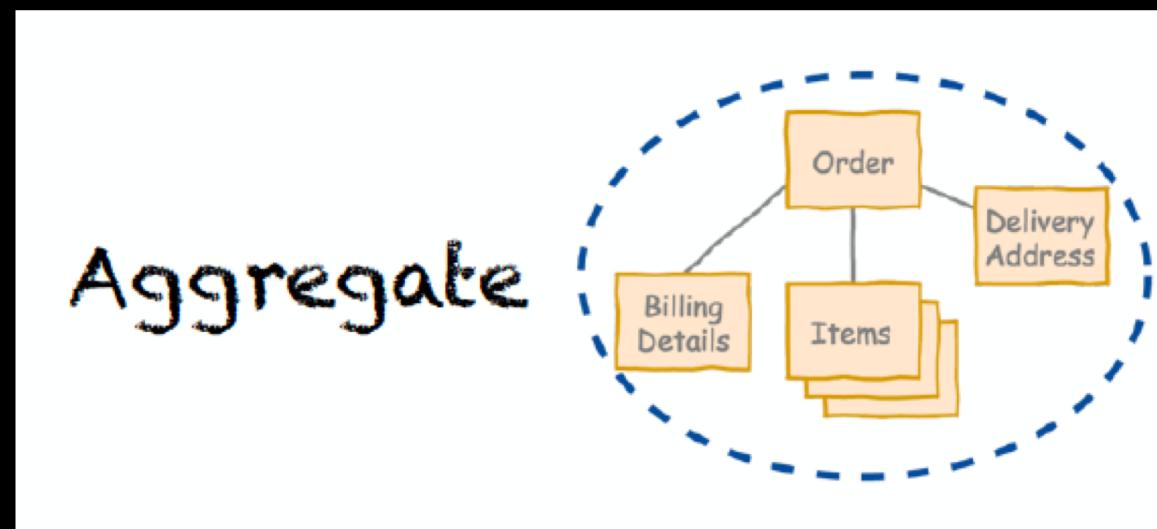
CQRS

DDD resumido (I)

- Un dominio es el conocimiento de negocio/actividad alrededor de lo que la aplicación resuelve (reglas de negocio).
- Sabiendo como se organiza mi empresa, defino un bounded context (agrupación de actividades que tiene sentido tratar como una unidad).
- En cada bounded context están claras las dependencias y contratos requeridos
- Buscamos un bajo acoplamiento con otros bounded context y una alta cohesión dentro de él.
- Dentro de cada bounded context tendremos un mismo lenguaje entre técnicos y personas de negocio. Este lenguaje llega incluso al código.



- **Value Object** -> Objectos inmutables que no tienen ciclo de vida ni entidad pero buenos para dar semántica al código e incorporar validaciones.
- **Aggregates** -> Unidades mínimas de cambio de estado o transacciones. Lo pueden componer varias entidades siendo la entidad aggregate root la responsable de mantener un estado consistente vía operaciones transaccionales (ACID)
- Hay una separación clara entre el dominio (que se hace) y la infraestructura (como se hace). Interfaces en los dominios que implementan en la infraestructura (desacoplamos dominio).



Los primeros pasos: CQS

Command Query Segregation

“Every method should either be a command that performs an action, or a query that returns data to the caller, but not both. In other words, asking a question should not change the answer. More formally, methods should return a value only if they are referentially transparent and hence possess no side effects.”

Bertrand Meyer (“*Object Oriented Software Construction*” (1988))

Dos tipos de métodos

Command

Cambian datos, nunca los retornan

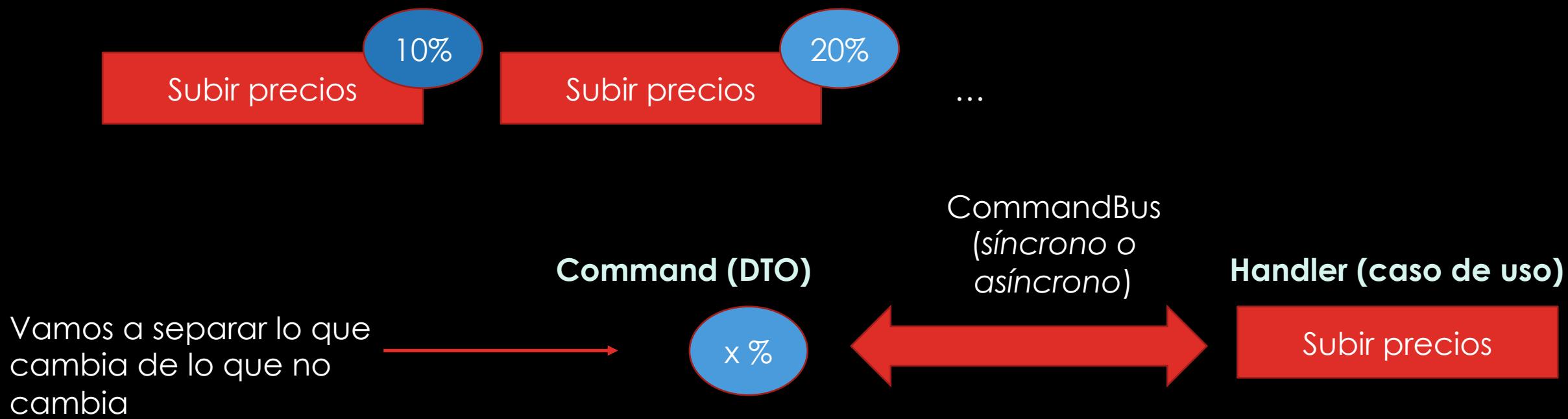
Query

Retornan datos, nunca los cambian

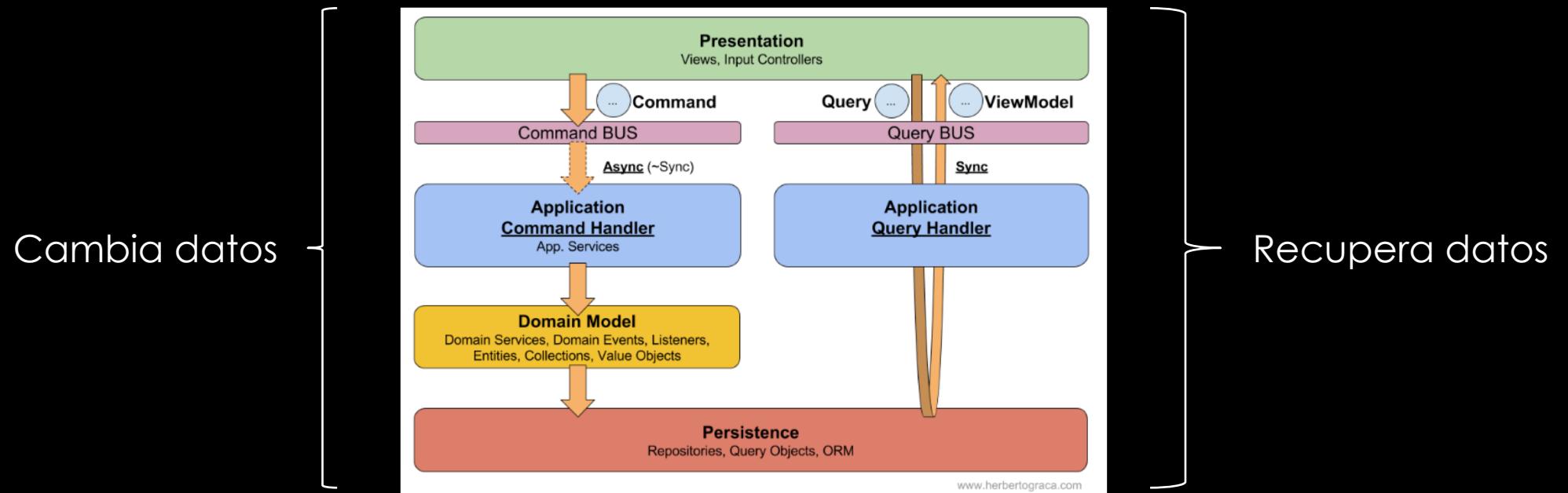
Patrón Command

- Nos permite ir a un enfoque donde nuestro dominio conoce los procesos de negocio.
- Encapsulamos todo lo necesario para ejecutar una acción o secuencia de ellas. Asíncrona o síncronamente.

Pero, ... ¿qué pasa si nuestro proceso siempre es el mismo pero cambian los parámetros de entrada?



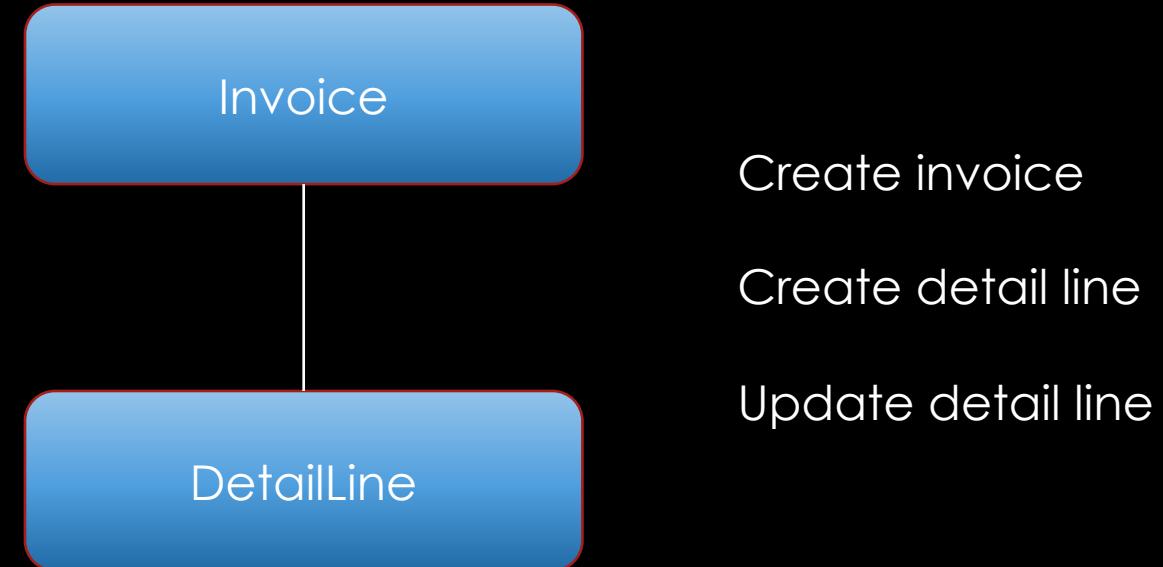
Juntando los conceptos de CQS, el nuevo Command y el CommandBus llegamos a él



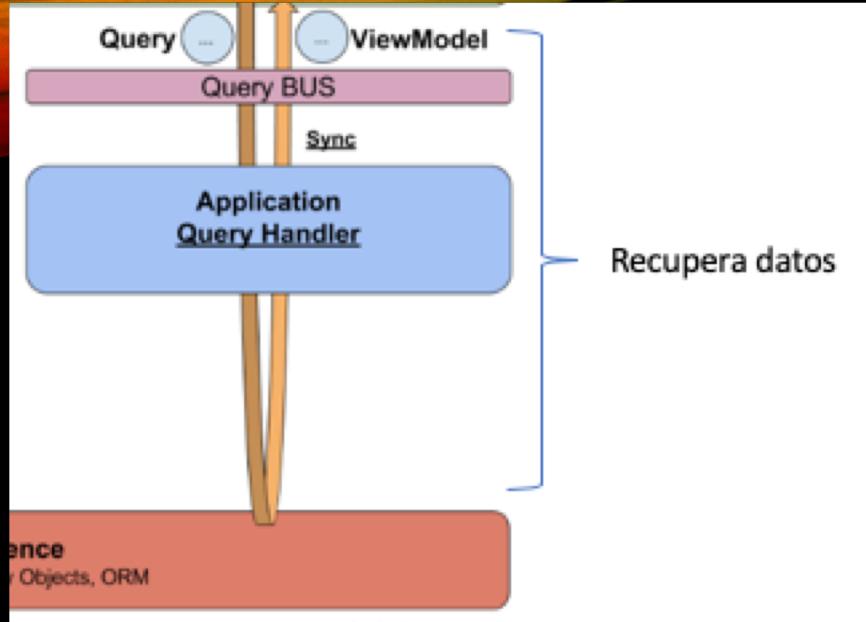
- Escritura y lectura separadas: podemos tratarlas por equipos diferentes
- Las segregación lleva a más piezas, pero más simples.
- Tenemos una orientación a comportamiento (el Command indica una tarea)

Demo:

CQRS y DDD con una única fuente de datos



Query (I)



No hay ejecución de procesos de negocio sobre los datos

- No necesito clases de negocio
- No necesito todos los datos de las entidades



Solo necesito **datos en crudo** que se muestran al usuario y **sólo** los que éste necesita

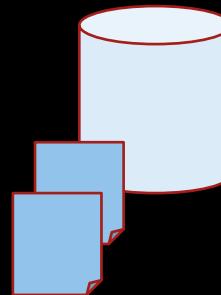
Query (II)

¿ Y si además separamos las fuentes de datos?



- Eliminamos restricciones de integridad de datos en la BBDD
- Uso de vistas (proyecciones) con los datos que necesito mostrar
- Ya no necesito un dominio
- La optimización de queries ya no es un problema
- En la parte del Command queda solo la consulta por agregado

¿ Y si pasamos de una BBDD RDBMS a una NoSQL?



- Usemos un MongoDB o Redis ya que no necesito el modelo relacional



Rendimiento

Query (III)

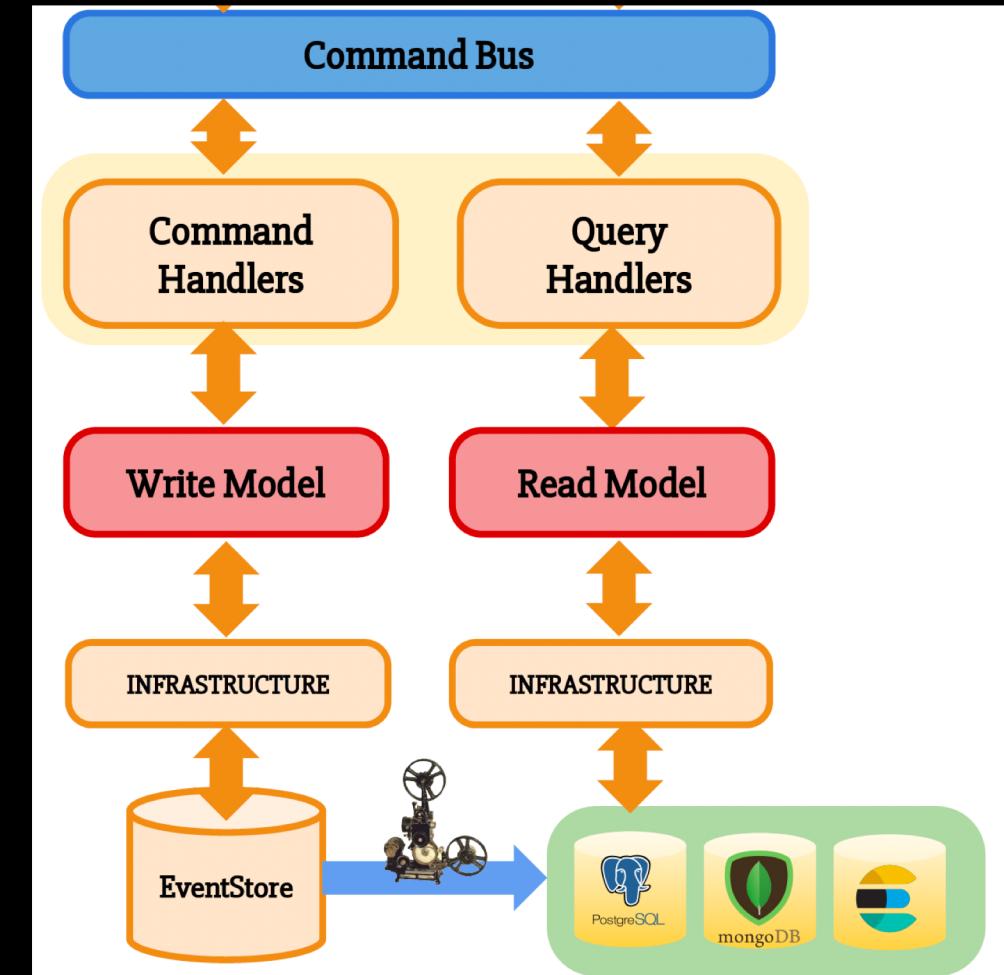
¿ Y como hacemos que la BBDD de la Query esté sincronizada con la del Command?

En las acciones de cambio del estado de mis CommandHandler lanza eventos que son escuchados por subscriptores (por ejemplo el que actualiza la BBDD lectura)

¿metemos eventos?



Inconsistencia eventual



Event Sourcing resumido (I)

Concepto:

Persistir el estado de un agregado en forma de secuencia ordenada de *eventos* que se habían aplicado al *agregado*.

Evento:

Mensaje que representa que algo ha pasado en el sistema (se nombran en pasado).

Event Store:

Lugar donde se almacenan los eventos aplicados sobre un agregado (identificado por un id). Sólo podemos añadir, no modificar ni borrar eventos.

Para hacer operaciones sobre un agregado debo hidratarlo (recuperar su estado actual aplicándole todos los eventos que le afectaron a este estado).

Event Sourcing resumido (II)

Versionado y upcasting:

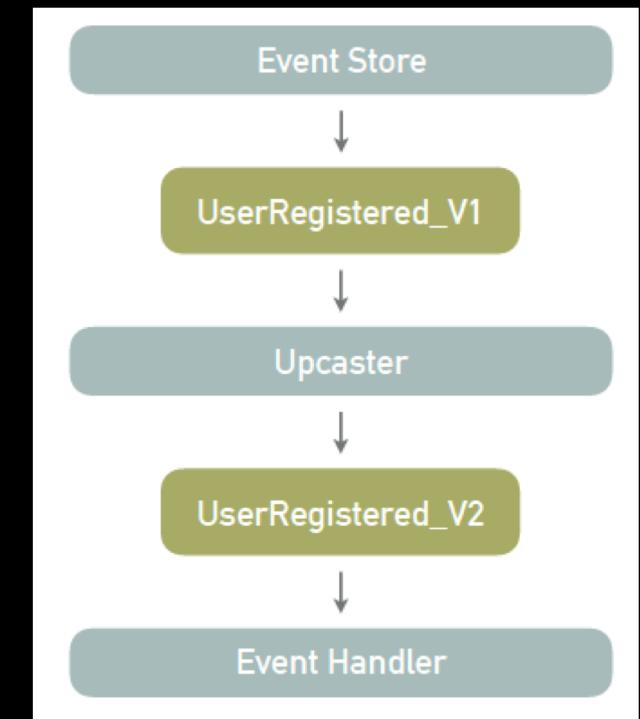
Ante nuevos requerimientos de negocio o refactor por deuda técnica puede ser necesario incorporar modificaciones a la estructura de nuestros eventos.



- Deprecamos campos en vez de eliminarlos
- Valores por defecto en campos nuevos
- No cambiamos el tipo de un evento, creamos uno nuevo



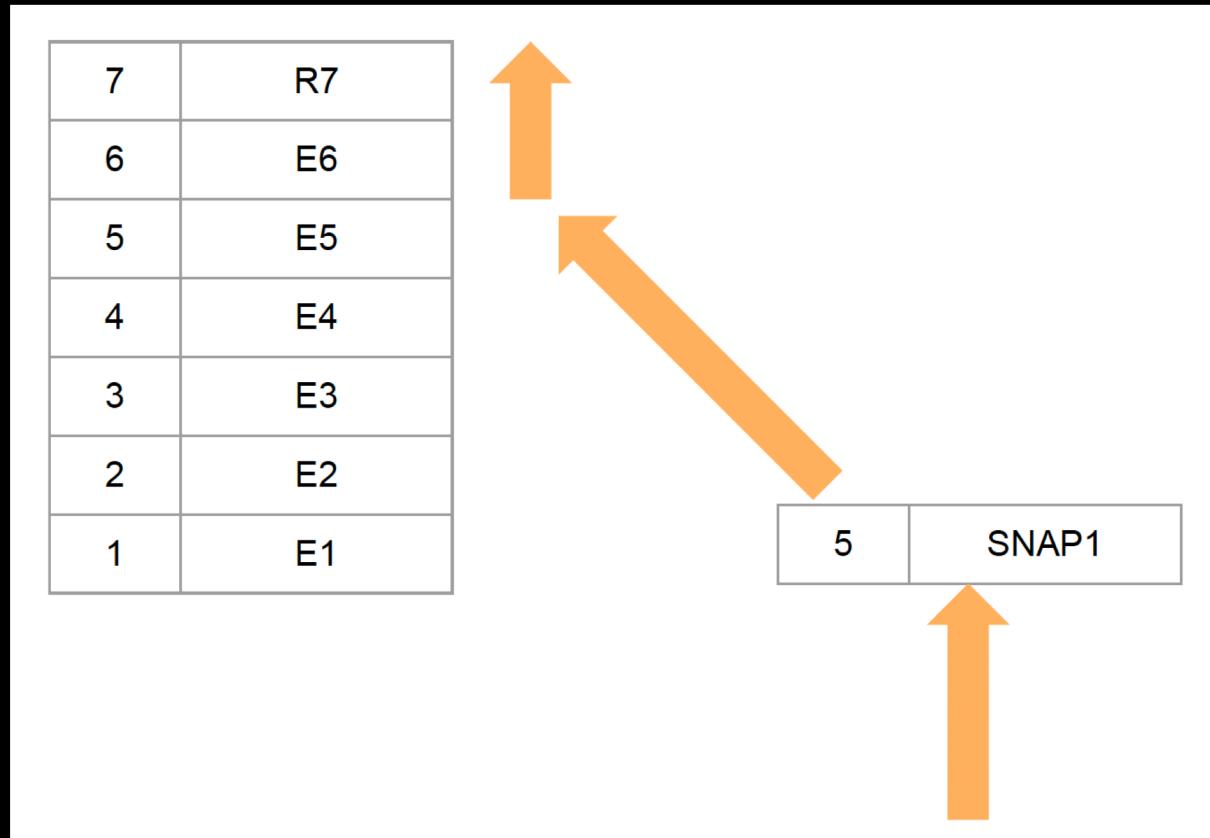
- Mas complejidad en el código
- Baja rendimiento
- Complejo en sistemas HA



Event Sourcing resumido (III)

Snapshot:

Evento que recoge el estado del agregado en un evento (indicado para historificación).



Mejoramos rendimiento

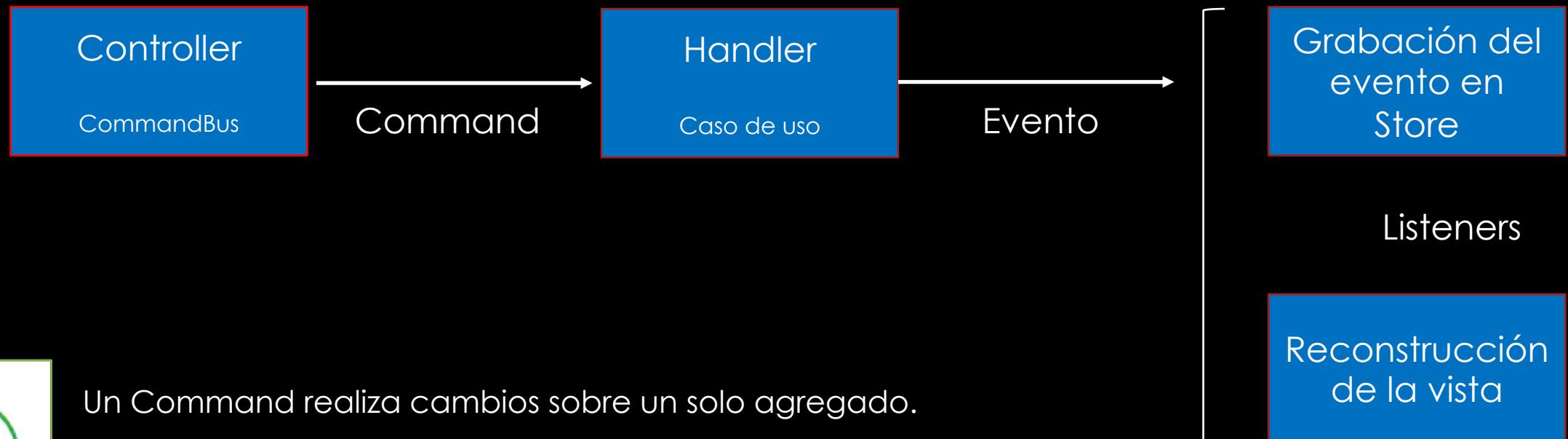
Event Sourcing: ventajas e inconvenientes

- La consistencia de datos en entornos de microservicios sin transacciones se soluciona.
- Para transacciones podemos usar Sagas
- Facilita depuración de errores y trae la auditoría de serie.
- Alto rendimiento. Unido a CQRS , podemos escalar de forma distinta lecturas y escrituras.



- No estamos acostumbrados.
- El evento muestra todo (incluidas inexperiencias del pasado).
- Ojo a los eventos duplicados y a la inconsistencia eventual (**awaitility** nos pueden ayudar).
- Tener una clara estrategia de upcasting, versionado y snapshot o tendremos problemas.
- El event store solo permite buscar por el id del agregado.

Flujo ante un Command

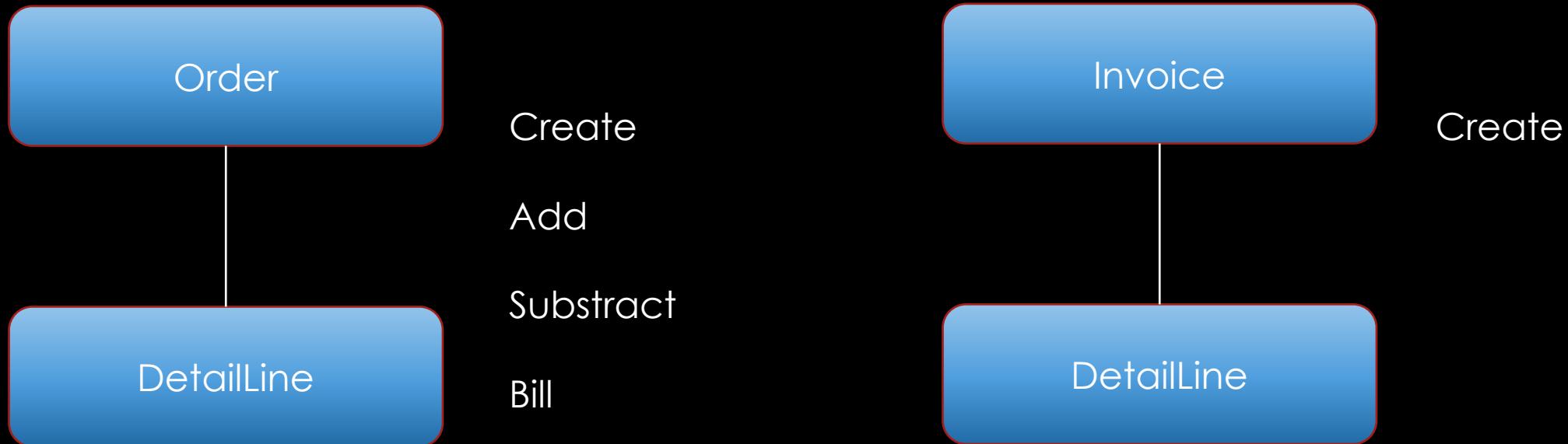


Un Command realiza cambios sobre un solo agregado.

Si debemos realizar cambios sobre otro, lanzaremos un eventos que al ser escuchado lanzará otro Command (respetamos SRP).

Demo:

CQRS, DDD y Event Sourcing con dos fuentes de datos



Frameworks CQRS+ES

Framework	Upcasting	Snapshots	Replaying
Broadway (PHP)	No (PR)	No (PR)	Not in core
Prooph (PHP)	MessageFactory	Yes, triggers on event count	Example code, off line
Axon (Java/Scala)	Upcaster / UpcasterChain	Yes, triggers on event count	Yes, ReplayingCluster
Akka Persistence (Java/Scala)	Event Adapter	Yes, decided by actor	Yes

Referencias:

- <https://martinfowler.com/bliki/CQRS.html>
- https://cqrss.files.wordpress.com/2010/11/cqrs_documents.pdf
- <http://udidahan.com/2009/12/09/clarified-cqrs/>
- <http://udidahan.com/2011/04/22/when-to-avoid-cqrs/>
- <https://airbrake.io/blog/software-design/domain-driven-design>
- <https://carlosbuenosvinos.com/event-sourcing-is-not-a-messaging-integration-pattern/>
- <https://github.com/CodelyTV/java-ddd-skeleton>
- <https://github.com/kotato/axon-examples>
- <https://pro.codely.tv/library/cqrs-y-event-sourcing-con-kotlin-y-axon-framework/67816/about/>