codely

DDD y CQRS: Preguntas Frecuentes

15 DE DICIEMBRE DE 2016 | SCREENCASTS

Hace unas semanas compartimos nuestra evolución desde código acoplado al framework a microservicios pasando por DDD. A raíz de esto, nos llegaron algunas dudas desde varios frentes: Desde los comentarios del vídeo en YouTube, hasta en el Slack de BcnEng (por cierto, aprovechamos para invitaros a uniros a él!), pasando por varios tweets. Con lo cuál, **en este vídeo vamos a intentar dar respuesta a esas dudas** que, en muchos casos, son las mismas dudas que nos surgieron a nosotros

al empezar con DDD y CQRS. Si no estáis muy metidos en ajo de CQRS y DDD, os recomendamos que empecéis por la introducción a Arquitectura Hexagonal que preparamos hace un tiempo. Esta Arquitectura de Software es el pilar en el que se basa DDD, y os podrá servir para empezar a indagar en estos temas :

Agradecer a todos los que nos habéis hecho llegar vuestras dudas. Esperamos poder aportar valor compartiendo nuestro punto de vista al respecto de:

Commands

Minuto 0:55. Importante primero destacar que estamos hablando de comandos entendidos dentro del mundo CQRS, y no del patrón de diseño Command. ¿Es posible dejar los **atributos públicos** para evitar tener que hacer getters? **Idealmente deberían garantizar su inmutabilidad**. Para ello deberíamos evitar exponer la escritura de sus atributos (hacerlos privados, y no tener setters si no sólo getters).

Command bus síncronos

Minuto 3:08. Partimos de la base de que **idealmente nuestro sistema debería ser tan asíncrono como podamos**. Eso deducimos de iniciativas como el Reactive Manifesto, y básicamente es la característica que hace que nuestros sistemas sean más escalables. No obstante, habrá ocasiones en las que **gestionar la consistencia eventual en la**

interfáz de usuario sea tremendamente complicado. Recordemos que desde el momento en el que procesamos de forma asíncrona nuestros comandos, nosotros no estamos devolviendo una respuesta definitiva ya que no sabemos el resultado que ese comando tendrá (si el registro de usuario irá bien, o no, por ejemplo). Con lo cuál, habrá ocasiones en las que **no nos compense que nuestros comandos sean procesados de forma asíncrona**. Básicamente porque la dificultad que implica hacer nuestros sistemas realmente reactivos y compatibles con la consistencia eventual, no nos compensa con el beneficio que obtenemos. Si nuestros sistemas no necesitan ese nivel de rendimiento en cuanto a escalabilidad, entendemos que es inasumible el coste que implica apostar por al asincronía. Por lo tanto, y por mucho que idealmente todo debería ser asíncrono, **es muy probable que tengamos command buses síncronos en nuestro sistema**. Estos buses sí bloquearán esperando la respuesta que le dé el CommandHandler para, por ejemplo, retornar desde el controlador de la API REST un HTTP 201 conforme el usuario ha sido registrado satisfactoriamente, o un 409 porque el identificador de usuario ya existe.
Conclusión: Hay que promover que los comandos sean asíncronos hasta donde podamos. Si no, tenemos otra implementación del CommandBus de forma síncrona para solucionar la papeleta 🙂

Identificadores UUIDs

Minuto 5. Aquí hablamos de los identificadores de tipo UUID. Este tipo de identificadores son **útiles para evitar delegar la responsabilidad de generación de IDs a nuestra infraestructura**. Dado que los UUIDs son generados de forma aleatoria, y con una probabilidad despreciable de colisión, podemos hacer que nuestras entidades tengan el identificador como uno de los parámetros necesarios para ser construidos. Con esto hacemos que no sea necesario pasar por el repositorio de base de datos para saber qué identificador tendrá la entidad, con lo que lo podemos esperar incluso desde fuera. Aquí el ejemplo de implementación de una entidad, y de cómo gracias a esto:

- El **test de ejemplo se simplifica mucho**
- **No hace falta que el cliente dependa de la respuesta del servidor para redirigir** a la página de edición tras la creación de una entidad. Como el cliente es quien ha generado el identificador, en el momento de recibir el HTTP 201 por parte de la API, puede redirigir (siempre que el comando sea síncrono tal y como comentábamos antes 🙂)

Desacoplar del framework

Minuto 7:30. En este punto planteamos los beneficios de desacoplarnos de la estructura de directorios marcada por los frameworks, y cómo podemos llegar a hacerlo. Tenéis el ejemplo de código del repositorio de CQRS y DDD para echarlo un ojo 😬. Importante el

detalle de separar lo que son las aplicaciones del código con la lógica de negocio #cosaFina👌.

Dónde publicar eventos de dominio

Minuto 11:55. Primero comentamos por qué preferimos la semántica de "record" frente a la que ofrecería el método "raise" para el momento de registro de los eventos de dominio. Este detalle es importante a pesar de lo sutil que podría parecer. Aquí es justamente donde radica la diferencia con la solución que proponía Carlos Buenosvinos en su post al respecto. Él plantea que es la propia entidad quien efectúa la publicación (a través de un servicio singleton). Lo que planteamos en el repositorio de ejemplo es que **quien registra el evento sí es la entidad, pero quien lo publica es el Aplication Service que representa el caso de uso**. Por lo tanto, de ahí la importancia en la semántica del método "record". Éste lo único que hace es guardar el evento en la entidad para que finalmente sí se publique. Todo esto es un detalle, pero es relevante desde el momento en el que, como explicábamos en la introducción a la Arquitectura Hexagonal, **lo que particulariza a los Application Service es representar un caso de uso de forma atómica**. Es decir, la responsabilidad de publicar el evento la debería tener el Application Service ya que es quién determina, en este caso, que el Vídeo ha sido creado y, por lo tanto, se puede publicar el

evento. Si movemos esa responsabilidad a la entidad dejamos de tener la posibilidad de evitar publicar el evento por circunstancias posteriores a la instanciación. Por ejemplo, que no se pueda persistir finalmente esa entidad. Además de los problemas a nivel conceptual, pensamos que dificulta el testing demasiado ya que nos obliga a usar costuras al acoplar la entidad al singleton que publica eventos.

Conclusión

Queremos aclarar que todo esto parte de nuestra interpretación al respecto de CQRS y DDD. Como decimos a lo largo del post, en muchos casos todo se basa en decidir si compensa o no asumir ciertos contras. Con lo cuál, habiendo explicado nuestra postura al respecto, nos encantaría escuchar diferentes alternativas a través de los comentarios del vídeo, en el Slack de BcnEng, o por Twitter 🙂. Por último, como dijimos en la charla, agradecer a todas esas personas que nos han ayudado a aprender todos estos conceptos. ¡¡¡Gracias!!!"

TAGS

Arquitectura Hexagonal DDD

Nivel avanzado PHP Testing

ANTERIOR
Estrenamos patrocinador oficial de CodelyTV: ¡IronHack!

SIGUIENTE
Learning process, faith, y a molar

¿Crees que puede interesar a más miembros del equipo?

- 💸 Descuento de hasta un 40%
- ✨ Gestión centralizada de cuentas
- 👷 Profesionales con amplia experiencia
- 💼 Facturas a nombre de empresa
- 📈 Reportes y analítica

Más información