**Next stepts**

1. retry system del snapshot.
2. **LIMPIAR EL COMMAND Y EVENT BUS, esto sirve para procesar los eventos con fecha pendientes. De esta forma se preserva el flujo de trabajos programados al momento de la reconstrucción.**
3. **Crear lo de .Notify y .Trace, esto, mas un menú para activar y desactivar del tracint (un simple bool, u otra instancia, que no haga nada y asi sea liviana para procesar miles de mensajes)**
4. **Probar el motor con 100 comandos, 1000 comandos y así sucesivametne, todos en el bus, para detectar problemas de performance. Todo esto se supone que sería asíncrono.**
5. Que limpie el command bus y el event bus y lo ponga todo en 0 los generadores de id en la base de datos.
6. Verificar que el log siempre escriba un único log por mensaje.
7. Hacer del snapshot un proceso secundario asíncrono de importancia media (como el tracer) que guarde una copia serializada del actor o aggregate (como un mensaje) en una tabla con connection string configurable (puede estar en otra base de datos con otro servior). El esquema podría ser EventStore.Snapshots. El event store rebuilder también puede reconstruir esto. Debe tener el mismo principio de append only. Al momento de guardar se puede disparar otro proceso secundario, como un IStaleSnapshotCollector que se encarga de mirar las tablas y borrar todas las versiones anteriores del actor y borrarlas. Proceso de importancia baja. Esto debe siempre poder ser compatible con al implementación anterior de snapshots en memoria. Si se implementa esto, entonces se podría devolver el temporizador de expiración de los actores en el cache, para liberar memoria que no esta usando. Esto seria una importante optimizadcion de performance, liberanto a los actores en memoria que no están haciendo nada…
8. Hacer que tracer se pueda apagar (un flag en un campo que si esta apagado, no hace nada) Esto no es tan importante.
9. El date time de NEventStore de Jonathan Oliver se ve muy bien. Quiza se pueda implementar.
10. El id del Command Bus y del Event Bus en la base de datos debería porder ser el id en long, para que pueda almacenar muchos, muchos.. muchos Ids…..
11. Que haya una politica de reintento en el worker. A medida que se vaya reintentando va a tardar mas en volver a reintentar, asi como se hace el poll del read model de parte del cliente. Configurable
12. Tambien la política de reintento del event handler, y ver la política de reintento del command handler, que sea incremental. Verificar si esto no es lo mismo que en el punto 1.
13. El TimerTaskFactory de Conference CQRS Journey se podría usar como un poll especifico para ver si se cumple una condición para asi saber que el evento se proceso correctamente. Esta muy buena la implementación. Se podría usar en paralelo a la implementación que hice yo.
14. El message log debería guardar, a parte del creation date (fecha donde se hizo insert) también la fecha de cuando se creo el mensaje (tanto para eventos como para comandos) para que se pueda filtrar del message log y asi restaurar un sistema hasta cierto punto.

Puntos de entrada para implementar un dominio con el Journey Framework:

1. Insertar el domain container en esta sección:
2. 

Arquitectura de un dominio. Assemblies = Layers

1. Domain: Aquí se encuentran los handlers y los aggregates
2. Domain.Commands: Aquí se encuentran los commands. Es el único lugar donde debería conectarse con la capa de aplicación.

Reglas:

1. Todo comando debe generar un evento. Asi se evita el reprocesamiento.
2. Entity Framework Query Optimization

martes, 7 de abril de 2015

11:43

En la capa de aplicación se podrian crear los commands. Quizás allí ejecutar las primeras validaciones de que todo esté bien.

La capa de aplicación contiene todos los objetos de de read model, por ejemplo, y tambien para enviar commands. Esto ayudaria a limpiar los controller, hacer que los command senders sean testeables y esten libres de inyecciones por todos lados, o sea, de interfaces que se repiten constantemente. Mejor es inyectarle una unico objeto de "Aplicación" (DDD).

El disable de entity framework para que sea más rápido.

public SimpleInventarioDao(Func<SimpleInventarioDbContext> contextFactory)

        {

            // performance optimizer.

            this.contextFactory = () =>

            {

                var context = contextFactory.Invoke();

                context.Configuration.AutoDetectChangesEnabled = false;

                return context;

            };

        }

Machine generated alternative text:
PROJECT 
ocøuG 
TEAM 
TEST 
ANALYZE 
public MerpContext 
public Database() 
this . Context 
Context get; private set; 
new Merpcontext(); 
this .Context. Configuration.AutoDetectChangesEnab1ed false; 
*region IDatabase Members 
public IQueryablecActivityType) ActivityTypes 
get return Context. ActivityTypes; 
public IQueryabIe«BusinesslJnit) BusinesslJnits 
get return Context. BusinessUnits; ) 
31:31 / 

Message Receiver.

Ahora se pueden ver como dos transcciones de vida larga en la base de datos a los procesos que miran las tablas del bus:



SOBRE PORQUE NO USO BUS EN MEMORIA – 3 RAZONES

1. Se pierde la capacidad de programar tareas (poniendo fecha de procesamiento del mensaje en el bus).
2. Se pierde la capacidad de ejecutar largos procesos de análisis de eventos. Si fuera en memoria se tendría que ejecutar todo primero antes de mostrarle cosas al usuario. Esto quizás si no fuera muy largo sería rapidísimo.
3. Es más difícil el multiusuario (cada vez que se hace commit ya puede un usuario ganar un turno en el event store)