CQELight.Docs Documentation

Hybrid Technologies Solutions

nov. 08, 2019

Introduction

1 Bienvenue sur la documentation **1**

1.1 CQELight : qu'est ce que c'est ? . . . 1
1.2 Getting started . . . 2
1.2.1 Le test du Hello World ! (event sourcé, évidemment . . .) . . . 2
1.3 Modélisation du domaine . . . 4
1.3.1 Généralités . . . 4
1.3.2 Architecture générale . . . 6
1.3.3 ValueObjects . . . 6
1.3.4 Du bon choix des types 6
1.3.5 Entities . . . 6
1.3.6 Aggregats . . . 7
1.3.7 Services . . . 7
1.3.8 Domain Events . . . 7
1.3.9 Factory et repository . . . 8
1.4 Commands et Queries . . . 8
1.4.1 Séparation CQRS . . . 8
1.4.2 Commands . . . 9
1.4.3 Queries . . . 10
1.4.4 En résumé . . . 11
1.5 Inversion of Control . . . 11
1.5.1 Généralités . . . 11
1.5.2 Enregistrement . . . 11
1.5.3 Résolution . . . 12
1.5.4 Spécificités . . . 13
1.5.5 Paramètres de résolutions . . . 13
1.6 Mapping modèle de données . . . 13
1.6.1 Création du mapping du modèle . . . 13
1.6.2 Utilisation des attributs . . . 14
1.7 Accès aux données . . . 15
1.7.1 Accès en lecture . . . 15
1.7.2 Accès en écriture . . . 16
1.7.3 Spécificités BDD relationnelle (SQL) . . . 16
1.7.4 Intégration dans un système CQRS . . . 17
1.8 Event Sourcing . . . 17
1.8.1 Généralités . . . 17
1.8.2 Modification du domaine . . . 17

1.8.3 Event store . . . 18
1.9 Configuration du dispatcher . . . 18
1.9.1 Généralités . . . 18
1.10 Bootstrapper . . . 20
1.10.1 Généralités . . . 20
1.11 Créer vos propres extensions . . . 21
1.11.1 Généralités . . . 21
1.11.2 Création de l'extension . . . 21
1.11.3 Spécificités de chaque type . . . 23
1.12 Tests unitaires et fonctionnels . . . 23
1.12.1 BaseUnitTestClass . . . 24
1.12.2 IoC . . . 25
1.12.3 Bus . . . 25
1.12.4 Test du dispatch . . . 25
1.12.5 Méthode d'extensions . . . 26
1.13 Scénario complet . . . 27
1.14 Modélisation du domaine . . . 27
1.14.1 ValueObject . . . 27
1.14.2 Du bon choix des types . . . 28
1.14.3 Entity . . . 29
1.14.4 Aggregate . . . 30
1.15 Commands . . . 32
1.16 Events . . . 34
1.17 Accès aux données . . . 36
1.17.1 Modèles . . . 36
1.17.2 Repository Json . . . 37
1.17.3 Changement des handlers . . . 38
1.18 Inversion of Control . . . 40
1.19 Queries . . . 43
1.20 Event sourcing . . . 45
1.20.1 Gestion de l'état de l'aggrégat . . . 47
1.20.2 Gestion d'un aggrégat event-sourcé . . . 48
1.20.3 Assemblage des élements . . . 48
1.21 Tests unitaires et fonctionnels . . . 49
1.21.1 Test unitaire . . . 49
1.21.2 Test d'intégration . . . 51
1.22 Finalisation . . . 52
1.23 IoC avec Autofac . . . 52
1.24 DAL avec EF Core . . . 53
1.25 Bus In-Memory . . . 54
1.26 Event sourcing avec EF Core . . . 55
1.26.1 Spécificités . . . 56

CHAPITRE 1

Bienvenue sur la documentation

La documentation du site se veut être interactive, participative et couvrir les différents projets et applicatifs qu'offre Hybrid Technologies Solutions. Vous y trouverez , en naviguant à l'aide du menu sur la gauche, plus d'informations sur l'utilisation de notre framework CQELight pour la conception d'application centralisées sur le métier, tout comme des conseils généraux pour la modélisation du domaine.

N'hésitez pas à parcourir la documentation ! En cas de manque, oubli ou erreur, vous pouvez créer une issue sur GitHub, nous la traiterons et procéderons à la mise à jour de la documentation.

1.1 CQELight : qu'est ce que c'est ?

La première question à se poser est : qu'est-ce qu'un logiciel ? Une réponse à cette question est :

> *Un logiciel est un outil permettant de résoudre des problèmes et d'ajouter un plus grand confort de travail pour un ou plusieurs métiers d'une entreprise*

De fait, si on considère cette définition, on retrouve un point très important, **la notion de métier**. En effet, généralement, les développeurs se focalisent sur l'aspect technique d'un logiciel, demandant à un product owner ou à d'autres personnes maîtrisant le métier, comment implémenter telle ou telle fonctionnalité (si ce n'a pas déjà été défini dans un cahier des charges) et tentent de faire rentrer le métier dans la technique qui a été mise en place.

Très souvent on constate qu'un logiciel qui a été créé il y a plus de deux ans devient plus lent, plus lourd, plus compliqué à maintenir et les adaptations à implémenter pour s'accorder à l'évolution du métier auquel il répond sont de plus en plus complexes et risquées. Le terme legacy est par ailleurs souvent utilisé par les développeurs pour déterminer ce genre de situation, et on arrive plus difficilement à trouver des personnes motivées pour en faire la maintenance.

CQELight n'est pas un outil magique qui fera que des foules se presseront pour faire de la maintenance sur l'applicatif qui sera construit avec lui. Cependant, il apportera un ensemble d'outils, de patterns et de bonnes pratiques permettant de simplifier, voire parfois totalement éliminer certaines parties de la maintenance nécessaire.

A l'instar de beaucoup de framework, CQELight ne contient que les briques de base permettant la construction d'un logiciel. C'est lui qui va se charger de la grande partie des problématiques infrastructurelles et architecturales pour que les développeurs puissent se focaliser sur l'implémentation du métier. L'avantage de cette vision des choses : si le logiciel est construit en se basant sur le métier au lieu de se focaliser sur la technique, il pourra plus facilement suivre

les évolutions du business, et même permettre à de nouveaux entrants sur le projet d'apprendre le métier en parcourant le code.

Quels sont donc les outils que CQELight met à disposition ? On peut en sortir une liste facilement, sachant que le concept de base est l'extensibilité et l'adaptabilité aux pratiques et outils existants :

— Gestion de la séparation du code en Command et Query
— Gestion de l'envoi/réception des Commands et Events
— Objets de base pour le modeling métier
— Configuration fine du comportement du système
— Gestion simplifiée de l'injection de dépendance
— Gestion simplifiée des accès aux données
— Maintenance évenementielle assistée

Il n'est pas impossible que certaines de ces notions ne vous parlent pas spécialement. Le but de cette documentation est de vous éclairer sur ces notions, et vous donner les informations pour les utiliser dans vos implémentations, avec l'aide de CQELight.

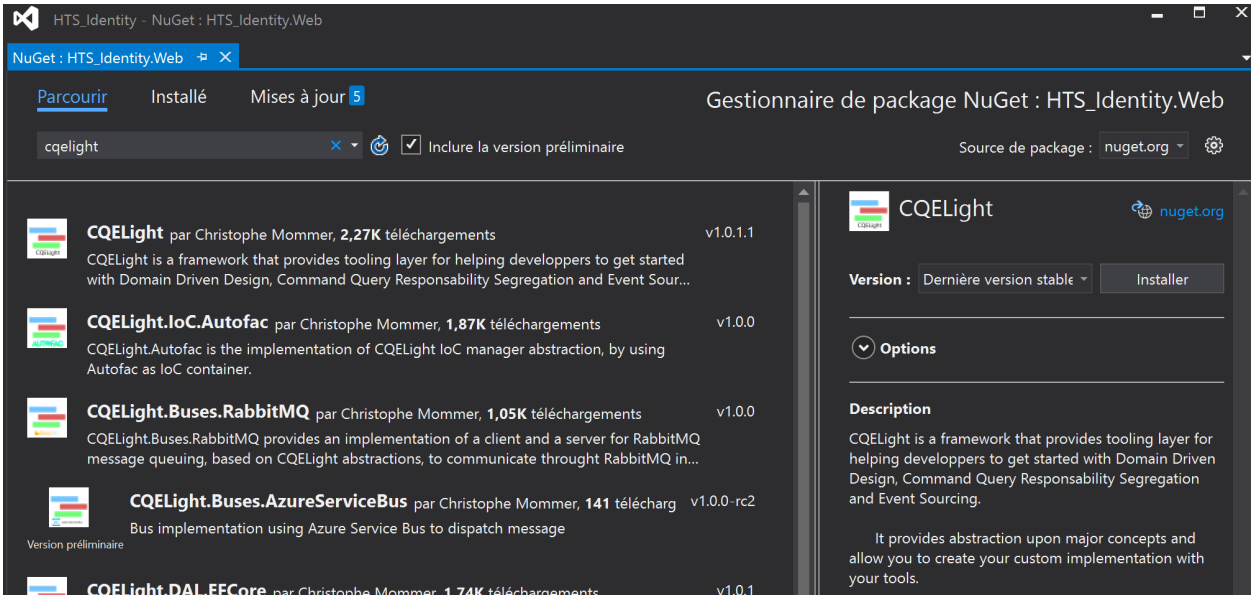
C'est parti, commençons par le *Getting started*, pour savoir comment débuter facilement et rapidement !

1.2 Getting started

1.2.1 Le test du Hello World ! (event sourcé, évidemment . . .)

Tout d'abord, il faut savoir qu'il y a des exemples disponibles sur notre repository GitHub, ici-même vous permettant d'approfondir selon les technologies que vous utilisez quotidiennement. Autre chose à savoir également, CQELight est distribué en packages NuGet modulaires, en .NET Standard 2.0 (compatible .NET Core 2.0+ et .NET Framework 4.6.1+).

Pour l'installer dans cet exemple, il faut commencer par créer un nouveau projet console (idéalement .NET Core) sous Visual Studio (ou l'éditeur de votre choix), et ensuite faire une recherche de package sur `CQELight`. Plusieurs résultats devraient apparaitre. Installer le package de base, `CQELight`.



Note : La dernière version stable peut avoir changé depuis la rédaction de cette documentation. Il est conseillé de

prendre toujours la dernière version stable pour avoir les dernières fonctionnalités et corrections de bugs.

Félicitations ! Vous venez d'installer CQELight sur un projet, nous pouvons dorénavant commencer à l'utiliser.

Créons maintenant deux nouveaux dossiers : `Events` et `Handlers`, ainsi que deux nouvelles classes : `GreetingsEvent` (dans le dossier Events) et `GreetingsEventHandler` (dans le dossier `Handlers`). Nous sommes prêt à commencer le code.

Il faut maintenant que dès lors que le système reçoit un event de type GreetingsEvent, le message "Hello World !" s'affiche sur la console. Faisons devenir notre GreetingsEvent un VRAI event, en héritant de la classe `BaseDomainEvent` dans le namespace `CQELight.Abstractions.Events` :

```
using CQELight.Abstractions.Events;
class GreetingsEvent : *BaseDomainEvent
{
}
```

Ensuite, faisons devenir GreetingsEventHandler un **VRAI** EventHandler, en le faisant implémenter l'interface `IDomainEventHandler` du namespace `CQELight.Abstractions.Events.Interfaces`, et implémentons le comportement désiré

```
using ConsoleApp1.Events;
using CQELight.Abstractions.Events.Interfaces;
using System;
using System.Threading.Tasks;
class GreetingsEventHandler : IDomainEventHandler<GreetingsEvent>
{
    public Task HandleAsync(
                    GreetingsEvent domainEvent,
                    IEventContext context = null)
    {
        Console.WriteLine("Hello world!");
        return Task.CompletedTask;
    }
}
```

Finalement, il est nécessaire de déclencher le mécanisme depuis le thread principal du programme. Pour ce faire, il faut publier l'événement dans le système. Etant donné que l'on fonctionne 100% In-Memory, il faut ajouter le package `CQELight.Buses.InMemory`.

Une fois fait, il faut configurer le système et envoyer l'événement :

```
using CQELight;
using CQELight.Dispatcher;
using HelloWorld.Events;
using System;
using System.Threading.Tasks;

namespace HelloWorld
{
    class Program
    {
        static async Task Main(string[] args)
        {
            new Bootstrapper()
                .UseInMemoryEventBus()
                .Bootstrapp();
```

(suite sur la page suivante)

(suite de la page précédente)

```
            await CoreDispatcher.PublishEventAsync(new GreetingsEvent());

            Console.Read(); // Used to avoid console to exit automatically
        }
    }
}
```

Note : Dans cet exemple, nous utilisons une version de Visual Studio permettant l'usage de C# 7.3, qui permet d'avoir une méthode `Main` d'une application console qui soit asynchrone. Si ce n'était pas votre cas, faites un appel avec `CoreDispatcher.PublishEventAsync().GetAwaiter().GetResult()` pour un résultat similaire.

Lancez le programme et voilà, vous avez publié votre premier event ! Il reste encore beaucoup de choses à découvrir avec CQELight, ceci n'est qu'un petit avant goût des possibilités offertes. Explorez les parties qui vous intéressent par le menu, ou jetez un coup d'oeil à notre repository pour savoir ce qui arrive.

Le code source de cet exemple est disponible à cette adresse.

1.3 Modélisation du domaine

1.3.1 Généralités

Pour modéliser votre domaine selon le Domain Driven Design (DDD dans la suite de la documentation), il est nécessaire de réfléchir à son découpage, afin de déterminer ce qui est essentiel à votre métier, et ce qui est annexe. Ce découpage se fait sous forme de Bounded Context (contexte borné).

Ce découpage en contextes se fait selon une **logique métier**. Il est nécessaire de faire ce cloisonnement, et de définir à l'intérieur de chaque contexte un **langage unifié**, l'Ubiquitous Language. Ce langage définit le nom des objets et des procédures qui évoluent au sein du contexte. Ce langage utilise la terminologie du métier, c'est à dire que les termes doivent être compréhensibles par une personne non technique dans l'équipe qui alimente le backlog fonctionnel, ou même par les clients/utilisateurs.

Le schéma général du DDD est le suivant :

Comme on peut le constater sur ce schéma, l'ensemble de la philosophie DDD est complète, concernant autant l'organisation que la technique, et peut paraître complexe. Mais il n'en est rien. Il faut partir de l'élément principal=Model-Driven-Design.

Avant toutes choses, il faut éclaircir le concept. Le système informatique qui va être créé sera piloté par un modèle, qui sera lui-même piloté par le domaine (le métier). Donc, le métier est au centre de la pensée. En partant de ce point de départ, on constate que le schéma se découpe en deux parties distinctes.

La partie basse se concentre sur l'organisation du code et de l'équipe. Beaucoup de concepts y sont abordés et dépassent le cadre de CQELight. Si vous désirez en savoir plus sur cette organisation, nous vous invitons très vivement à vous tourner vers le net et la littérature sur le DDD, assez conséquente, ou vous tourner vers notre e-formation approfondie sur le DDD (à venir). La seule information a récupérer de ce bloc est la notion de séparation en "Bounded Context". Ici, chez Hybrid Technologies Solutions, pour marquer cette séparation, nous aimons faire une solution Visual Studio par contexte traité, afin d'être sûr de garantir le cloisonnement et l'indépendance.

Par contre, nous allons nous attarder sur la partie haute, car il s'agit à proprement parlé de la modélisation technique du domaine et ce qui tourne autour.


DOMAIN EVENTS
access with
REPOSITORIES
SERVICES
access with
express change with
express model with
ENTITIES
encapsulate with
act as root of
express identity with
encapsulate with
AGGREGATES
express state &
computation with
MODEL-DRIVEN DESIGN
VALUE OBJECTS
encapsulate with
model gives structure to
isolate domain expressions with
encapsulate with
define model within
LAYERED ARCHITECTURE
encapsulate with
FACTORIES
UBIQUITOUS LANGUAGE
cultivate rich model with
keep model unified by
CONTINUOUS INTEGRATION
SHARED KERNEL
SHARED KERNEL
names enter
BOUNDED CONTEXT
CORE DOMAIN
interdependent contexts form
overlap allied contexts through
relate allied contexts as
CUSTOMER /SUPPLIER
work in autonomous, clean
assess/overview relationships with
CONTEXT MAP
minimize translation
CONFORMIST
avoid overinvesting in
GENERIC SUBDOMAINS
segregate the conceptual messes
support multiple clients through
free teams to go
OPEN HOST SERVICE
translate and insulate unilaterally with
loosely couple contexts through
formalize
BIG BALL OF MUD
ANTI-CORRUPTION LAYER
SEPARATE WAYS
PUBLISHED LANGUAGE

1.3.2 Architecture générale

Il est nécessaire d'architecturer un projet en couche ("Layered architecture") dans le schéma. C'est un concept très répandu, qui veut que le code soit découpé en plusieurs couches, une couche ne connaissant que la couche inférieure à elle, et exposant des informations à celle au-dessus d'elle, sans avoir connaissance de cette dernière. Il existe plusieurs architectures en couches, nous ne nous étendrons pas sur le sujet, mais chez Hybrid Technologies Solutions, on préconise une architecture décentralisée et découpée selon le mode CQRS, avec une encapsulation forte. Nos exemples montrent ce mode de fonctionnement, et vous pouvez également avoir différentes approches dans notre e-formation (à venir).

La couche qui nous intéresse ici est la couche dite domaine (ou business). C'est celle qui contient les objets représentant le métier et permettant de faire fonctionner le logiciel en adéquation avec les besoins métiers. Comme on peut le voir sur le schéma, il est nécessaire de découper un contexte en plusieurs entités ("Entities" sur le schéma). Ces entités représentent des blocs cohérents et consistants au sein d'un contexte. Elles véhiculent des données mais également un comportement en rapport avec ces données.

Cependant, une entité n'est pas la seule et unique représentation du contexte. On va également y trouver des ValueObjects, qui transportent une donnée immuable, au sens métier. Un ValueObject comporte une notion métier très forte.

1.3.3 ValueObjects

La notion de ValueObject permet de transporter une valeur métier forte, qui est identifiée par l'unicité de chacun de ses membres à un niveau métier. Par exemple, si on modélise un système bancaire et qu'on veut utiliser la notion d'argent, on créera une classe qui permet d'encapsuler le type primitif C# `decimal`, et d'y ajouter, si nécessaire, des informations supplémentaires (par exemple la devise).

Si on teste l'unicité d'un tel objet, on ne va pas le faire au niveau technique/système (comparaison de référence objet), mais on va le faire avec un point de vue métier. Dans notre précédent exemple, c'est l'égalité du montant et de la devise qui détermine si les deux sont égaux, et non les pointeurs vers la mémoire.

Un ValueObject n'est pas qu'un simple type complexe qui transporte des informations de façon immuable, il peut également transporter une ou plusieurs actions. Toujours d'après notre exemple, on pourrait ajouter dans notre ValueObject la possibilité de faire une addition. Le point important sera de garantir l'immuabilité afin d'éviter les effets de bord de son utilisation.

1.3.4 Du bon choix des types . . .

Lors du design d'un système, on tend naturellement à utiliser des types primitifs pour définir les valeurs de nos objets (`string`, `int`, `DateTime`, `...`). Cela est bien, cependant, ils ne véhiculent aucune information métier forte ni n'assurent de sécurité à la compilation (par exemple, comment distinguer un string qui réprésente le nom de celui du prénom à la compilation ?).

Cet « effet de bord » nous force à repenser la définition des types de notre domaine en encapsulant les types primitifs dans des objets qui véhiculent non seulement une identité forte, mais assurent aussi une sécurité à la compilation ainsi qu'au développement (en ajoutant des vérifications à la construction par exemple).

1.3.5 Entities

Une entité est un objet plus complexe, qui transporte des données et véhicule un comportement. Contrairement à un ValueObject, les données d'une entité sont muables (peuvent être modifiées), mais uniquement par les comportements internes qu'elle aura prévu. Ainsi, on évitera de laisser toutes les propriétés en visibilité publique (on favorisera la portée private ou internal), et on autorisera la modification de ces données que par le biais de fonctions bien définies qui feront les vérifications nécessaires.

La notion d'entité n'a véritablement pas de sens en tant qu'unité, elle doit faire partie d'un tout. On ne traite pas avec une entité de façon isolée, sinon, c'est qu'il s'agit d'un aggregat et non d'une entité. Concernant les données, celles-ci peuvent être diverses (ValueObject, autres entités, types primitifs) mais ne peuvent pas contenir une instance d'un aggregat (ni l'aggregat propriétaire, ni un autre aggregat du contexte).

Enfin, afin d'accentuer le métier, certains développeurs préfèreront mettre le constructeur privé et utiliser une méthode factory qui sera statique et qui portera le nommage propre au domaine afin de créer une nouvelle entité. Concernant l'implémentation comportementale et les données, il faudra coller au maximum aux contraintes du métier. Il est aussi possible d'utiliser la factory si la logique de construction est trop complexe pour être passée par un simple constructeur.

1.3.6 Aggregats

L'aggregat est le point d'entrée dans le métier afin de permettre une gestion cloisonnée du domaine. Si on est dans un système event-sourcé et CQRS, c'est également lui qui répondra aux Commands et propagera des Events. Il affiche donc une API publique et est responsable de la cohésion de son état interne. Il ne faut pas permettre de modifications de son contenu depuis l'extérieur car on pourrait arriver à un état inconsistant. Il faut considérer un aggregat comme un regroupement logique d'élements métiers définis au préalable.

Note : Attention cependant car il n'est pas nécessaire d'implémenter obligatoirement les autres types d'objets par aggregat. Ce qui est impérativement nécessaire par contre, c'est de garder toute mutation du domaine sous contrôle des fonctions de l'aggregat.

Note : Il n'est également pas grave d'avoir de multiples aggregats pour un contexte donné, si tant est que cela corresponde au besoin métier. De la même façon, il est préférable d'avoir une finesse de plusieurs petits aggregats (principe SOLID S) que de mettre tout au sein d'un seul et d'embarquer des données et un comportement inapproprié (souvent avec effet de bords).

1.3.7 Services

La notion de services permet de définir des comportements qui ne nécessitent pas d'état mais véhiculent une notion métier. Dans le cas de notre exemple, on mettrait en service un système de conversion d'argent d'une monnaie vers une autre, car il y a un comportement métier fort, totalement indépendant et potentiellement complexe.

La définition d'un comportement au sein d'un service ou d'un objet métier (aggregat, entity ou ValueObject) reste soumise à l'appréciation du besoin métier. La règle principale c'est de savoir s'il est nécessaire d'avoir un état pour effectuer l'opération. Toujours avec l'exemple d'un système de change, si la notion de change est directement dans le système a été modélisé, l'opération aura un sens d'être implémentée dans un objet du domaine. A contrario, si l'information est fournie au moment de la transformation et n'est pas conservée car ce n'est pas le but du domaine, on choisira d'en faire un service.

1.3.8 Domain Events

Les événements sont une réaction du domaine à une sollicitation, notamment une modification. Ils sont importants même si le système n'utilise pas forcément une technologie de type Event-Sourcing. On peut utiliser un événement au lieu d'un type de retour car il transporte plus d'informations métier. De la même façon, il peut correspondre au résultat d'un appel sans forcément avoir besoin d'un système de type messaging.

L'événement domaine est une des notions les plus importantes dans l'ensemble du système car on véhicule un changement métier fort. Une des façons de modéliser un domaine peut par exemple être de rechercher l'ensemble des

événements qui peuvent être générés (Event-Storming) et les découper conceptuellement pour les regrouper sous une même logique.

1.3.9 Factory et repository

Factory et repository sont deux patterns bien connus des développeurs. Ces deux patterns permettent la récupération d'objets métiers depuis une source de données externe (repository) ou depuis des données en mémoire (factory). On utilisera ces derniers si l'instanciation de nos objets est une procédure métier plus complexe qu'une simple initialisation (par exemple affectation de valeurs calculées ou génération d'objets).

L'implémentation de ces patterns reste à l'appréciation de chacun. Il n'est pas nécessaire d'avoir un objet pour pouvoir l'utiliser (parfois une simple méthode statique suffit). On aura d'ailleurs plutôt tendance à utiliser une factory pour les implémentations métier complexes afin de ne pas créer de couplage fort avec une source de données quelconque.

1.4 Commands et Queries

1.4.1 Séparation CQRS

Nous avons vu comment modéliser le domaine dans l'article *Modélisation du domaine*. Lorsque que nous devons penser, au niveau domaine, à l'API publique, on pense généralement aux méthodes (et propriétés accessibles éventuelles) de l'aggregat.

En respectant le pattern CQRS, il faut faire une distinction entre les interactions avec le système provoquant une modification (commands) et les lectures de données (queries). Le pattern stipule que l'exécution d'une query doit se comporter comme une fonction mathématique pure, à savoir qu'un appel identique doit fournir un résultat identique s'il n'y a pas eu d'altération du système entre temps.

De l'autre côté, une commande va modifier le système, à savoir que chaque appel va mettre le système dans un état différent. On considère généralement qu'une query a un type de retour selon ce qu'on veut requêter et une commande retourne `void` (ou `Task` pour les commandes asynchrones).

Si l'on utilise le système événementiel (soit en event sourcing, soit en event driving), la réaction de notre aggregat à une commande sera de créer un ou plusieurs événements et de les ajouter dans sa collection interne. L'appelant sera alors chargé de regarder dans cette collection pour voir le résultat de son appel.

Si on n'utilise pas le système d'évènement métier, on va alors utiliser une notion de résultat d'appel. Ce retour doit contenir la valeur métier de l'échec ou de la réussite. CQELight expose une classe permettant d'encapsuler ce retour, la classe `Result`. Cette classe de résultat peut être utilisée au niveau de l'API publique que donne l'aggregat (sans utilisation des évènements) ou dans les objects dépendants de l'aggregat (entités et valueObjects). Cela permet d'encapsuler la valeur de retour et de garder la signature de notre méthode honnête (en stipulant que l'action peut avoir un résultat qui peut être positif ou négatif selon les cas).

Note : De façon générale, lorsqu'on implémente le code du domaine, il est préférable d'éviter d'utiliser les exceptions car le système de gestion des exceptions est beaucoup plus lourd et coûteux que celui des valeurs de retours. De même, dans une logique de programmation, on doit garder les exceptions pour des cas exceptionnels et non pour des résultats métiers attendus.

La classe `Result` se présente sous deux formes : une avec valeur et une sans valeur. On ne distingue le fait, que ça soit un échec ou succès, que par le flag `IsSucces`. Cette classe permet d'éviter l'utilisation d'un type primitif tel que le booléen qui ne transporte pas assez d'informations métier.

```
public enum TransferCannotBeDoneBecause
    {
        NotEnoughMoney
    }

public Result TransferMoney(Account from, Accout to, Amount amount)
{
    // Business logic
    if(!from.HasEnoughMoney(amount))
    {
        return Result.Fail(TransferCannotBeDoneBecause.NotEnoughMoney);
    }
    return Result.Ok();
}
```

Note : La classe `Result` de base ne transporte pas d'autres informations qu'un booléen de succès ou d'échec. Elle doit idéalement servir de base à toute forme de résultat dans notre domaine. CQELight fourni cette classe de base ainsi que le class `Result<T>` qui permet d'encapsuler une information de retour. Vous restez libre de créer vos propres héritages de `Result` selon les besoins de vos domaines (dans l'exemple ci-dessus, on aurait pu créer une classe `TransfertResult` qui hérite de `Result` pour le cas précédent).

1.4.2 Commands

Afin de rendre tout ceci possible, il est nécessaire d'exposer publiquement notre aggregat (pour que chacun puisse l'appeler selon son besoin), mais également de fournir une méthode de récupération d'un aggregat totalement reconstitué. L'utilisation d'une factory statique au niveau de l'aggregat reste possible, ce n'est pas forcément totalement optimal si le domaine est complexe (surtout qu'on arrive à une pollution du domaine avec des problématiques infrastructurelles). De la même manière, cela implique de laisser notre aggregat avec une visibilité publique générale, ainsi que potentiellement les objets dont il est composé, ce qui peut créer des problèmes de séparations en couches.

La notion de commande asynchrone de CQELight a été créée à cet effet. Elle est totalement dispensable dans les cas où vous devez interagir de façon synchrone avec votre domaine. Cependant, même si la logique d'instanciation/récupération de votre domaine est peu complexe, il est recommandé d'utiliser le pattern asynchrone de CQELight pour maximiser les performances et s'assurer d'un vrai découpage CQRS. Cela permet également de garder les objets domaines visibles uniquement à un niveau internal, évitant les raccourcis malheureux.

Une commande est la volonté d'un intervenant extérieur à intéragir avec le domaine concerné. Il s'agit donc d'un simple DTO, qui véhicule les informations nécessaires pour exécuter le code requis, et par son type/nom, envoie également l'information de l'action à effectuer. De façon générale, l'handler de la commande va considérer que s'il en récupère une, elle est considérée comme valide (il ne faut donc pas faire de vérification métier des paramètres de la commande dans le handler, on aurait une fuite des contrôles du domaine). La commande est responsable de ses données, il est recommandé de créer un constructeur avec vérification des paramètres.

Note : Attention, il n'est question ici que de vérification technique des paramètres (null, empty strings, ...), et non pas métier. Généralement, c'est le domaine qui va réagir en fonction des valeurs, mais un premier filtre peut déjà être effectué si nécessaire, éviter d'avoir à polluer notre domaine avec des contrôles techniques.

Une fois la commande créée, il est nécessaire de l'envoyer dans le système afin que le(s) handler(s) disponible(s) puisse(nt) y réagir. Pour effectuer cette action, on utilisera une instance du dispatcher qui se chargera de l'envoyer dans les bus nécessaires. Le rôle du handler est de s'occuper des problématiques d'infrastructure pour restaurer une instance d'aggregat (depuis la base de données, avec des événements, appel à une factory, ...).

Note : Même si on parlé de plusieurs handlers, il est fortement recommandé qu'il n'existe qu'un seul handler pour une commande donnée, et ce afin d'éviter plusieurs comportements inattendus (race-condition, deadlocks, accès concurrents, ...)

De même lorsque l'aggregat est restauré et que l'action du domaine est invoquée, il y a fort à parier qu'un résultat a été produit (événement, information de retour, ...). Le rôle du handler de commande sera également de s'occuper du traitement de ce retour (par exemple envoi des évènements par le biais du dispatcher). Encore une fois, si vous avez besoin d'un appel synchrone au domaine, mieux vaut se passer de ce fonctionnement asynchrone.

```
using CQELight.Abstractions.CQS.Interfaces;
public class ExecuteDomainAction : ICommand
{
    //Some properties

    public ExecuteDomainAction()
    {
        //Execute some parameters checking here
    }
}
```

```
using CQELight.Abstractions.CQS.Interfaces;
public class ExecuteDomainActionHandler : ICommandHandler<ExecuteDomainAction>
{

    public Task HandleAsync(ExecuteDomainAction command, ICommandContext context =␣
↪null);
    {
        //Retrieve an instanciated aggregate

        //Execute domain action

        //Treat result of domain action
    }
}
```

Note : Il est préférable d'éviter que nos handlers renvoient des exceptions car les bus n'ont peut-être pas de mécanisme traitement des exceptions particuliers, ce qui peut causer un crash ou une instabilité globale du système, voire une perte de l'information d'échec, menant à un comportement inattendu. Il est fortement recommandé d'éviter toute forme d'exception dans ces appels et traitements et d'encapsuler les traitements (récupération comme exécution niveau domaine) par des try-catch pour éviter ce genre de déconvenues.

1.4.3 Queries

A l'inverse de la commande qui est une volonté d'interagir avec le domaine et de le modifier, les queries permettent une récupération d'informations qui auront été générées par le domaine. Dans un logiciel de gestion classique, la majorité du temps passé à interagir avec la source de données se fera en lecture plutôt qu'en écriture. Ici, le concept de CQRS qui propose de séparer en deux piles différentes les lectures et les écritures prend tout son sens car le développeur restera libre d'implémenter différemment la pile des lectures pour l'optimiser.

De la même façon, les logiciels de gestion se contentent très rarement de travailler exclusivement avec une source de données volatile type mémoire vive, il y a toujours une forme de persistance. Lorsqu'on décide de persister les données, il faut garder en tête le pattern de fonctionnement pour de stocker les données de façon à ce que la lecture

soit optimisée et indolore (quitte à dénormaliser à l'extrême) plutôt que d'essayer d'optimiser le stockage, ce qui ralentira les temps de traitements.

1.4.4 En résumé

Pour résumer, à un niveau aggregat, le pattern CQRS impose une distinction entre récupération de données et modification du système (Command Query Separation), tandis qu'à un niveau système, les commandes seront utilisées pour interagir globalement avec le domaine. Les handlers se chargent des problématiques globales d'infrastructures, laissant ainsi le domaine pur. Finalement, les queries permettent de récupérer des données qui auront été stockées de façon optimisée, afin de permettre un affichage optimal.

1.5 Inversion of Control

1.5.1 Généralités

L'IoC (inversion of control) est une pratique de développement logiciel consistant à coder uniquement avec des abstractions et utiliser un container afin de récupérer la bonne implémentation selon le contexte. Ceci permet au code métier d'être hautement extensible (on change une implémentation sans changer la logique), testable (on peut définir nos implémentations de test pour piloter un comportement) et plus facilement maintenable (les abstractions sont clairement séparées des implémentations).

Ce concept est souvent couplé à l'injection de dépendance (Dependency Injection) qui consiste à stipuler que les dépendances abstraites doivent être fournies à la construction par un mécanisme d'injection.

De ce fait, CQELight permet l'utilisation de l'IoC et l'injection de dépendance dans son système, bien que cela ne soit pas obligatoire pour l'ensemble du système. Les avantages cités ci-dessus s'appliquent également dans le cas présent, et certaines extensions font un usage intensif de l'IoC, comme beaucoup de systèmes modernes (par exemple la configuration Asp.net Core).

1.5.2 Enregistrement

L'enregistrement est, de façon générale, géré lors de l'appel à la méthode `Bootstrapp()` de l'extension IoC. Il faudra alors que l'extension IoC se charge de récupérer les enregistrements du système pour les traiter selon ses spécificités. Le cas particulier est l'enregistrement de résolution lors de la création d'une extension, si vous voulez profiter de la puissance offerte d'un container Ioc, de vouloir faire un enregistrement. Dans ce cas précis, vous ne pourrez utiliser que les types d'enregistrements offerts par le système et non les spécificités du container utilisé.

Pour ce cas particulier, il faut passer par une collection interne au bootstrapper. Ce dernier fourni un point d'entrée simplifié : `AddIoCRegistration`. Cet appel doit être fait dans la méthode d'extension du bootstrapper (pour en savoir plus, lisez la documentation sur l'*Créer vos propres extensions*).

Il y a trois façon d'enregistrer dans le bootstrapper : par type, par instance et par factory. La différence réside dans le mode de résolution. Un enregistrement par type donnera à chaque résolution une nouvelle instance, un enregistrement par instance donnera l'instance qui a été enregistrée (singleton) et un enregistrement par factory permettra d'exécuter une logique de création/récupération personnalisée (invoquée à chaque résolution). Si cela s'avère insuffisant, il est toujours possible d'utiliser les méthodes natives du container par le biais de la méthode de bootstrapping.

Note : Les plugins officiels CQELight d'IoC permettent de configurer le container à l'aide des outils offerts par ce dernier en plus des types de CQELight. Il est fortement conseillé aux créateurs de plugins IoC d'en faire de même

---

```
// Register by type - Need implementation type and corresponding abstract types
bootstrapper.AddIoCRegistration(new TypeRegistration(typeof(InMemoryCommandBus),␣
↪typeof(ICommandBus), typeof(InMemoryCommandBus)));

// Register by instance - Need the instance and corresponding abstract types
bootstrapper.AddIoCRegistration(new InstanceTypeRegistration(configuration,␣
↪typeof(InMemoryEventBusConfiguration)));

// Register by factory - Need a lambda and corresponding abstract types
bootstrapper.AddIoCRegistration(new FactoryRegistration(() => efRepoType.
↪CreateInstance(dbContext),
                                dataUpdateRepoType, databaseRepoType,␣
↪dataReaderRepoType));
```

Note : Attention cependant, si aucune extension d'IoC n'a été configurée, vos enregistrements seront faits en vain. Bien que l'IoC soit fortement recommandé, il n'est pas obligatoire, il est préférable de toujours garder une possibilité hors IoC, même si cette dernière est fortement limitée.

1.5.3 Résolution

De base, l'injection de dépendances est faite par le biais des constructeurs. Vous pouvez, dès lors que vous avez activé l'utilisation d'une extension IoC, passer vos abstractions dans les constructeurs (de vos handlers d'events ou commands par exemple), qui seront automatiquement résolues par le système sans que vous vous en préoccupiez.

Il s'agit de la méthode de récupération des objets depuis le container la plus recommandée. Cependant, il est possible de faire des résolutions manuelles. A cet effet, il est prévu une notion de scope. Un objet résolu n'est garanti valide que dans le cadre d'un scope donné. Si scope est terminé, il est possible que l'objet résolu ne soit plus dans un état consistant.

CQELight fourni une API pour la résolution à n'importe quel moment de votre code, autre que le constructeur. Il y a deux façons de récupérer un scope de résolution : l'API statique (`DIManager`) ou l'utilisation d'un `IScopeFactory`. Un `IScopeFactory` étant un type abstrait, il est nécessaire de l'avoir en dépendance dans le constructeur

```
using(var scope = DIManager.BeginScope())
{
    var implementation = scope.Resolve<IAbstraction>();
}

public MyClass(IScopeFactory scopeFactory) // Ctor
{
    using(var scope = scopeFactory.BeginScope())
    {
        var implementation = scope.Resolve<IAbstraction>();
    }
}
```

Note : Attention à la durée de vie. La majorité des containers IoC en .NET gèrent eux-mêmes la durée de vie des objets qu'ils ont résolus. De fait, dans l'exemple ci-dessus, si `IAbstraction` est un `IDisposable`, l'appel de la méthode Dispose sera faite en même temps que celle du scope.

Comme souvent, la méthode d'instance est fortement recommandée si vous en avez la possibilité. Il peut arriver que

parfois il soit nécessaire de passer par l'API statique (méthode statique, pas de possibilité de modifier le constructeur, impossible de se faire injecter un type dans le constructeur, ...).

L'utilisation de l'API du `DIManager` est conditionnée à l'appel de la méthode `DIManager.Init()` qui prends en paramètre un `IScopeFactory`. Généralement, cet appel est réalisé par les plugins d'IoC de CQELight. Si vous développez un plugin pour un container IoC, pensez à faire cet appel au bootstrapp de votre extension.

1.5.4 Spécificités

1.5.5 Paramètres de résolutions

Généralement, une résolution est faite sans nécessité de préciser des paramètres particuliers. Il arrive cependant que certains types aient besoin d'un ou plusieurs paramètres pour que la résolution se fasse (si ces paramètres sont dynamiques à l'exécution). Pour les paramètres que le container IoC connait, la majorité de ces derniers arrivent à les gérer sans aide. Par contre, il peut arriver qu'il y ait besoin de paramètres spécifiques non résolvables.

Pour gérer ces derniers, il y a deux façons de préciser un paramètre lors de sa résolution : par nom ou par type. S'il n'y a qu'un paramètre spécifique, ou plusieurs dont le type est différent, la résolution par type est possible (et recommandée). Si ce n'est pas possible (par exemple deux paramètres de type string), alors la résolution par nom entre en jeu.

Pour résoudre un objet en précisant un paramètre par son type, il faut faire l'appel de la façon suivante :

```
using(var scope = _scopeFactory.GetScope())
{
    var instance = scope.Resolve(new TypeResolverParameter(typeof(string), "value"));
}
```

Pour résoudre un objet en précisant un paramètre par son nom, il faut faire l'appel de la façon suivante :

```
using(var scope = _scopeFactory.GetScope())
{
    var instance = scope.Resolve<IAbstraction>(new NameResolverParameter("param1",
↪"value"));
}
```

Note : Attention, certains providers IoC ne supporte pas nativement ce comportement particulier (comme par exemple `Microsoft.Extensions.DependencyInjection`. Vérifiez que votre provider le supporte ou vous risquez d'avoir une exception à l'exécution.

1.6 Mapping modèle de données

1.6.1 Création du mapping du modèle

Avant de pouvoir utiliser nos modèles avec un repository, il faut créer une couche de mapping, qui sera ultérieurement utilisée dans les plugins pour savoir comment gérer les entités dans une source de données. Pour cela, CQELight mets à disposition un ensemble d'attributs à appliquer à vos classes de données. Il reste possible de persister directement les objets du domaine dans un repository, mais en le faisant sans passer par un modèle dédié intermédiaire, il pourrait y avoir une pollution du domaine avec des problématiques de persistance (constructeur sans paramètre, visibilité publique, ...).

Pour éviter ce problème, il a été créé trois objets permettant de faire le lien entre une source de données et nos objets domaines : `PersistableEntity`, `ComposedKeyPersistableEntity` et `CustomKeyPersistableEntity`. Ces trois classes abstraites héritent de la classe abstraite de base `BasePersistableEntity` qui contient les élements communs, à savoir les valeurs de modification et de suppression. A un niveau d'abstraction supérieur, on retrouvera l'interface globale commune, `IPersistableEntity`, qui permet à l'équipe de développement de créer ses propres entités persistables.

La création du modèle consiste a créer l'ensemble des objets qui seront persistés dans la source de données choisies. Il n'y a pas beaucoup de règles à suivre, si ce n'est de suivre les recommandations du provider d'accès aux données qui aura été choisi par le biais du plugin.

1.6.2 Utilisation des attributs

Cependant, afin d'offrir une certaine flexibilité dans la création de ce modèle, il faut permettre une certaine personnalisation du modèle, comme par exemple le nom de la colonne ou de la table qui sera utilisée en cas d'utilisation de SGBD relationnel. Tous ces attributs sont disponibles dans le namespace `CQELight.DAL.Attributes`.

Note : Bien que certains ORM comme Entity Framework offrent déjà des attributs pour faire le mapping des modèles, nous avons souhaité redéfinir les notre afin de ne pas dépendre d'un provider particulier. Ainsi, il peut sembler parfois qu'il y ait un « double emploi », mais c'est afin de permettre de réaliser une seule fois le mapping et d'être compatible avec n'importe quel provider. Une attention particulière est donc portée à l'attention de l'équipe réalisant les mappings d'utiliser les attributs de CQELight (ou des assemblies qui sont communes, comme `System.ComponentModel.Annotations`) au lieu d'attributs spécifiques d'un provider, et ce pour éviter de se retrouver bloqué sur une seule technologie de persistance.

Les attributs disponibles pour la création du modèle sont :

- **TableAttribute** : permet de définir le nom de la table dans lequel l'entité doit être stockée. Cet attribut permet de spécifier un nom de table et un nom de schema (spécificité SQL Server).
- **ColumnAttribute** : permet de définir le nom de la colonne dans laquelle la propriété de la donnée doit être stockée. Cet attribut permet de spécifier le nom de la colonne.
- **PrimaryKeyAttribute** : permet de définir sur une propriété d'une entité laquelle sert de clé primaire (valeur d'identification unique et unitaire d'une entité). Cet attribut permet de spécifier le nom de la colonne clé.
- **ComposedKeyAttribute** : permet de définir sur une entité l'ensemble des propriétés définissant la clé primaire composée (valeur d'identification unique composée de l'unicité de l'ensemble des propriétés choisies). Cet attribut permet de spécifier le nom des propriétés à utiliser pour la définition de la clé composée.
- **IgnoreAttribute** : permet d'ignorer une propriété dans l'élaboration du modèle.

Les attributs disponibles pour l'optimisation du SGBD sont les suivants :

- **IndexAttribute** : permet de définir un index sur une propriété. Cet attribut permet de définir un nom d'index et le fait que l'index doit respecter une clause d'unicité.
- **ComplexIndexAttribute** : permet de définir un index composé sur plusieurs propriété. Cet attribut permet de définir le nom de l'index et le fait que l'index doit respecter une clause d'unicité.
- **NotNaviguableAttribute** : permet de définir les propriétés qui ne doivent pas être parcourue lors du traitement en profondeur. Certains ORM (comme Entity Framework) parcoure systématiquement la grappe d'objets pour définir leur état. Cet attribut permet de bloquer ce parcours.

Les attributs disponibles pour la gestion des relations sont les suivants :

- **ForeignKeyAttribute** : permet de définir sur une propriété « objet » qu'il s'agit d'une clé étrangère. Cet attribut permet de définir le nom de la propriété dans l'objet distant (en cas d'existance de plusieurs relations) et permet aussi de définir le comportement à suivre en cas de suppression.
- **KeyStorageOfAttribute** : permet de définir qu'une propriété heberge la valeur clé étrange d'un objet défini par un attribut ForeignKey. Cet attribut permet de prendre en paramètre le nom de la propriété objet clé étrangère.

Note : La gestion des collections doit se faire idéalement dans les deux sens, à savoir une propriété de départ et une propriété d'arrivée (pour le 1-1 ou le 1-Many) et deux propriétés : une pour l'objet afin de naviguer et une (ou plus

en cas de clé composée) pour la valeur clé de l'objet. Le type a utiliser pour les collections dans le 1-Many doit être de type ICollection<T> et ce afin d'être générique avec la totalité des providers disponibles sur le marché. La simple existence d'une propriété ICollection détermine l'existence d'une relation 1-Many. Il faut dès lors utiliser les attributs ci-dessus sur l'objet « maître » de la relation.

Note : Au jour d'aujourd'hui, il n'y a pas la possibilité de faire nativement une relation Many-to-Many, il est nécessaire d'utiliser un objet de transition.

Si le plugin du provider que vous avez choisi le supporte, vous pouvez également utiliser des attributs généraux issus du framework .NET (comme par exemple MaxLengthAttribute). Afin de savoir si c'est supporté, rendez-vous sur la documentation du plugin en question. Les implémentations de provider DAL doivent à minima supporter les attributs énoncés ci-dessus. Si vous avez besoin d'un exemple de mapping qui couvre l'intégralité des cas ci-dessus, rendez-vous sur la classe contenant les entités utilisées pour les tests unitaires sur le provider DAL Entity Framework Core.

1.7 Accès aux données

Dans les applications de gestion modernes, il est devenu obligatoire d'avoir accès une source de données durable (base de données, fichiers, ...). Cette obligation a donné lieu à la naissance de beaucoup d'outils divers et variés, dont par exemple les ORM. Cependant, il arrive parfois qu'au cours de la vie d'un logiciel, le type de source de données soit amenés à évoluer (passage de stockage sous forme de fichier en BDD par exemple).

A cet effet, il est impératif de commencer à penser son code pour que ce changement soit le plus anodin possible. Il faut créer une couche d'abstraction au dessus de l'accès au données, en utilisant le pattern **repository**. L'utilisation de ce pattern est repris dans le schéma de base du DDD, car l'accès aux données fait partie intégrante d'un logiciel métier.

CQELight fourni les abstractions de base d'un repository. En travaillant uniquement avec ces dernières dans votre code, vous serez libre, en un changement de ligne dans votre bootstrapper, de changer de source de données.

Avertissement : Il est tentant d'utiliser uniquement les abstractions et implémentations de base pour procéder à la gestion des données. En procédant de la sorte, vous vous évitez probablement du travail mais vous perdez en **testabilité** (les abstractions de bases sont testables uniquement dans une certaine limite), vous perdez en **visibilité** (les opérations utiliseront une terminologie technique et non métier) et vous perdez en **performance** (en utilisant les méthodes de base, vous n'avez pas la main sur toute l'API d'un ORM particulier par exemple, chose que vous pouvez optimisez en créant vos propres méthodes)

Nous conseillons de créer un repository par concept domaine persistable qui hérite de l'implémentation de base du repository de la technologie que vous avez choisi. Certes, cela oblige à redéfinir certaines choses en cas de changement de solution de persistence, mais vous vous évitez les problèmes cités précédemment.

1.7.1 Accès en lecture

Note : Nous avons volontairement fait le choix de rendre les APIs de lecture uniquement asynchrone, car cela dépend d'une source externe dans la quasi-totalité des cas.

La lecture des données dans votre source est probablement l'opération que vous ferez le plus souvent. Il y a énormément de technique d'optimisation à ce niveau pour gagner en performance et en temps de traitement (cache, optimisation requêtage, désactivation du suivi des modifications, ...).

Afin de permettre la consultation, nous fournissons l'interface `IDataReaderRepository<T>`. Cette interface expose deux méthodes de lecture : `GetByIdAsync` et `GetAsync`. La récupération par identifiant permet de lire un élement uniquement sur la base de son identité, tandis que le `Get` permet de récupérer une collection répondant à certains critères.

La méthode `GetAsync` permet de spécifier en paramètre :

- Un filtre sous forme de prédicat auxquels les élements devront répondre afin d'être dans la collection de résultat
- Un ordre particulier, qui sera effectué côté serveur
- Un flag indiquant si on récupère les élements qui ont été marqués comme supprimés de façon logique
- Le ou les objets/collections liés à charger lors de la récupération. Cette option est utilisée dans le cadre de relation entre entités, et est donc reservée de façon quasi-exclusive aux SGDB relationnels

Cette méthode renvoie un `IAsyncEnumerable` qui peut-être itérée ou transformée de façon asynchrone, permettant une récupération et une évaluation des paramètres lors de la demande de récupération des données.

1.7.2 Accès en écriture

Pour avoir des données à lire, il faut d'abord en écrire. L'interface qui permet d'écrire les données est un peu plus complète, car elle offre une finesse de distinction entre l'ajout et la modification. La majorité des méthodes de cette inferface permettent donc d'appliquer un marquage sur les entités afin que lorsque la transaction sera marquée comme complétée (par le biais de la méthode `SaveAsync`), l'opération soit réalisée de façon atomique selon ce qui a été décidé auparavant.

Note : L'appel à `SaveAsync` permet aux classes repository enfants de gérer la complétion de sa propre transaction métier (pattern Unit of Work), chose qui n'aurait pas été aisément réalisable si les méthodes Insert ou Update avaient fait l'enregistrement directement.

Chaque méthode d'écriture propose une version unitaire et une version multiple (ex : `MarkForInsert` et `MarkForInsertRange`). La suppression est également possible uniquement par le biais de l'ID, et ce afin d'éviter à avoir à procéder à un chargement de l'entité pour uniquement la supprimer. La gestion de la suppression permet l'utilisation d'un mode physique (la ligne est supprimée en base de données) ou d'un mode logique (la ligne est modifiée avec un flag qui l'indique comme supprimé). En cas de suppression logique, on peut indiquer lors de nos `GetAsync` si l'on veut remonter les enregistrements ou non, chose impossible en fonctionnement physique.

Note : Dans un fonctionnement CQRS-EventSourcing, les données remontées par les repositories seront des transformations d'évènements optimisés pour la lecture, l'utilisation de la suppression logique est contre productif car les vues ne sont pas la source de vérité. Il faudra bien penser à activer la suppression physique, désactivée par défaut, durant les appels. Il est possible de définir ce comportement par défaut lors des développements de vos plugins et d'ignorer les paramètres des fonctions. Les plugins CQELight officiels donnent la possibilité de préciser ce comportement lors du bootstrapp.

1.7.3 Spécificités BDD relationnelle (SQL)

Malgré que le nombre de source de données soit conséquent, le monde des bases de données relationnelles ne peut pas être ignorés. A cet effet, une interface dédiée à ce type de source a été ajoutée afin de permettre d'utiliser leurs spécificités (exécution de code SQL). De façon générale, il est fortement recommandé de ne pas exécuter du code SQL directement dans le code applicatif mais de passer par des méthodes de transformation. Certains cas, cependant, peuvent nécessiter d'utiliser l'API SQL directement. Il suffira d'utiliser l'interface `ISqlRepository`.

L'interface `ISqlRepository` fourni les méthodes à cet effet, permet l'utilisation du SQL directement sur la base de données. Les méthodes ne permettent pas de récupération de collection de données, uniquement de faire une mo-

dification ou de récupérer une valeur scalaire unitaire, ceci afin de décourager l'utilisation de ces APIs de façon trop régulière.

1.7.4 Intégration dans un système CQRS

Le pattern repository ainsi que les abstractions (et les implémentations fournies) sont suffisantes pour faire un système fonctionnel. Cependant, dans le cadre de la méthodologie CQRS, il est préférable de créer une couche Query, qui utilise les repository afin d'obtenir les données, et d'utiliser une système de cache.

Si vous utilisez également les domain-events (avec ou sans Event Sourcing), il est également conseillé de faire de l'invalidation de cache à l'aide des évènements. Tous ces concepts sont avancés et sont expliqués et fournis à titre d'exemple dans les documentations associés ainsi que les exemples disponibles sur GitHub.

1.8 Event Sourcing

1.8.1 Généralités

En event-sourcing, on considère les évenèments comme l'unique source de vérité. De ce fait, chacun des évènement est sauvegardé de façon durable, afin d'être réutilisé ultérieurement, notamment pour la prise de décision.

En effet, de façon très classique en développement informatique, il arrive assez souvent qu'on récupère des informations depuis une source de données afin de donner tout ce qu'il faut pour que l'utilisateur puisse effectuer une action en toute connaissance de cause. Généralement, on aura créé une base de données qui représente un instantané d'une situation, et on ignore (ou tout du moins on ne connait pas en détails) comment on est arrivé à cette représentation (même s'il arrive parfois qu'on créé des tables historiques ou qu'on profite de la flexibilité de certaines fonctionnalités de la base pour sauvegarder les modifications de données).

Cependant, en suivant la logique CQRS, la base de données doit être optimisée pour la lecture, quitte à dupliquer de la donnée. On risque dès lors de se retrouver avec du bruit, des informations non nécessaires, ou pire encore, des données qui n'ont pas été rafraichies et ne sont plus pertinentes, sur lesquelles l'utilisateur prendrait une décision qui ne serait, de fait, pas pertinente. Il serait également nécessaire de récupérer les données depuis de multiples sources, rendant possiblement le système contre performant. La seule solution pour prendre une bonne décision est de recréer l'état dans lequel était le système en prenant compte tout ce qui s'est passé. On appelle ceci **la réhydratation à base d'évènements**.

1.8.2 Modification du domaine

CQELight fourni des outils pour faciliter ce processus, notamment la notion d'event store. Comme son nom l'indique, un event store permet de stocker les évènements, avec une gestion tant en écriture qu'en lecture et une automatisation des routines. Il s'agit d'une extension, il vous faudra dès lors installer le provider qui correspond à votre stack technologique.

Bien que le comportement soit spécifiquement implémenté dans les extensions, certains concepts sont communs. Tout d'abord, pour commencer, les évènements viennent réhydrater un aggregat, de la même façon que c'est lui qui génère les évènements. Sauf que l'ensemble des développeurs qui veulent utiliser le DDD pour modéliser leur domaine n'a peut-être pas envie de faire un système event-sourcé. A cet effet, il faut explicitement définir son aggregat comme étant utilisé dans un système event-sourcé, en héritant de la classe `EventSourcedAggregate<T>`

```
public class MyEventSourcedAggregate : EventSourcedAggregate
{
    // Aggregate implementation
}
```

Note : Il n'y a pas beaucoup de différences entre un `AggregateRoot` et un `EventSourcedAggregate`, l'essence fonctionnelle reste la même. La seule différence réside dans le fait que l'aggregat doit pouvoir exporter un état sérialisé. Attention, cela ne veut en aucun cas dire que l'état doit être public, il suffit juste de pouvoir l'exporter de façon sérialisée afin de le sauvegarder dans l'event store si nécessaire.

L'aggregat doit également avoir un état qui doit pouvoir être muté selon les évènements qui sont arrivés. Cette notion d'état est quelque chose qui existe déjà en DDD, mais qui doit être approfondi en event-sourcing. A cet effet, une classe de base, `AggregateState`, disponible dans le namespace `CQELight.Abstractions.DDD` mets à disposition les premiers éléments pour permettre la réhydratation, à savoir la possibilité d'ajouter les callback d'application de évènements pour muter l'état, et la possibilité de se sérialiser. Bien entendu, comme une grand majorité des choses dans CQELight, ces méthodes peuvent être overridées par vos implémentations si cela s'avère nécessaire.

1.8.3 Event store

CQELight propose des extensions implémentant des event store selon les abstractions fournies dans l'assembly de base. Les abstractions à implémenter se trouve dans le namespace `CQELight.Abstractions.EventStore.Interfaces`

- `IEventStore` : C'est l'interface principale, le coeur du système d'event sourcing. L'event store doit définir les fonctions de récupération et de lecture des évènements tout comme la fonction d'écriture. A noter qu'une implémentation de cette interface est suffisante pour faire un système event sourcé où tout serait géré à la main, sans l'automatisation de CQELight, en gardant des objets métiers standards.
- `IAggregateEventStore` : Il s'agit d'une interface permettant de récupérer de façon plus automatisée les aggregats event sourcé totalement réhydratés. Les implémentations prennent en charge les problématiques de réhydratation, comme par exemple l'utilisation d'un snapshot comme base de travail.
- `ISnapshotBehavior` : Interface de contrat permettant de gérer la notion de snapshot pour des raisons de perfomances et de stockage.

CQELight mets à disposition deux providers d'event store qui proposent des implémentations pour ces abstractions, CQELight.EventStore.EFCore et CQELight.EventStore.MongoDb.

1.9 Configuration du dispatcher

1.9.1 Généralités

Le dispatcher est le chef d'orchestre du système, permettant de délivrer les informations en faisant le lien entre la demande d'envoi et le bus de destination, voire parfois même le destinataire, tout en fournissant une API simplifiée et accessible. Il existe deux versions du dispatcher : une API statique, le `CoreDispatcher`, et une API d'instance, implémentation de l'interface `IDispatcher`.

CQELight fourni une implémentation de cette interface (utilisée par le `CoreDispatcher`) : le `BaseDispatcher`. Il est recommandé, de façon générale, d'utiliser la version d'instance plutôt que la version statique, pour des contraintes d'accès concurrentiels et de performances. Cela rends également le code plus explicite en marquant le dispatcher comme étant une dépendance nécessaire au fonctionnement d'une classe donnée.

Bien qu'il soit recommandé d'utilisé la version d'instance, dans certains projets (comme les applications desktop), il y a quelques avantages d'utiliser en plus la version statique. En effet, le `CoreDispatcher` permet également de stocker des références vers certains parties du système, dans le processus en cours. Ainsi, on pourra lui demander d'avoir une référence vers un objet donné, pour capter des évènements, dans le contexte courant, grâce à la méthode `AddHandlerToDispatcher`, qui prends une instance dérivant de la classe object en paramètre (donc fondamentalement n'importe quel type système). Cet objet doit être un `IDomainEventHandler`, `ICommandHandler` ou `IMessageHandler` afin d'être ajouté et éligible lors de l'envoi.

Le `CoreDispatcher` est également le seul à pouvoir transmettre des messages applicatifs, qui implémentent l'interface `IMessage`. Ces messages sont souvent utilisés dans des contextes MVVM (WPF/Xamarin), exclusivement in-process, afin de découpler les interactions en lien entre View et ViewModel.

Etant donné le rôle central qu'a le dispatcher, il faut qu'il puisse être configuré finement afin d'être sûr que chaque envoi d'informations dans le système arrivent bien à destination. Il est possible de fournir une configuration, à l'aide du fluent builder de configuration, le `DispatcherConfigurationBuilder`.

Note : En l'absence de configuration, le dispatcher utilise la configuration par défaut, qui consiste à envoyer chaque évènement/commande à chaque bus qui a été défini dans le bootstrapper, sans aucune gestion d'erreur, sérialisés en JSON.

En sélectionnant un type spécifique, ou un ensemble de type (par le biais du namespace par exemple), on peut appliquer des choix comportementaux. Les élements configurables sont :

- L'envoi sur un ou plusieurs bus. Cela permet de définir par exemple, quels évènements sont des évènements internes au contexte, et lesquels doivent être publiés extérieurement.
- L'utilisation d'un moteur de sérialisation. Cela est nécessaire si le transport est particulier, auquel cas, le bus récupèrera l'instance du moteur de sérialisation et pourra l'utiliser si nécessaire.
- Un callback de gestion des erreurs s'il y a une ou plusieurs exceptions. Ce callback récupère l'exception rencontrée et permet de définir un traitement.
- La définition si le ou les type(s) choisi(s) est/sont « SecurityCritical », qui permet de définir si c'est un clone de l'instance qui est envoyé aux custom callback, ou si c'est l'instance réelle (ouvrant une porte à une modification des propriétés par un custom callback).

Il faut donc appeler le `ConfigurationBuilder` afin de pouvoir définir le comportement à adopter

```
var builder = new DispatcherConfigurationBuilder();

builder
    .ForAllEvents()
    .UseAllAvailableBuses()
    .HandleErrorWith(e => { Console.WriteLine(e); })
    .IsSecurityCritical()
    .SerializeWith<JsonDispatcherSerializer>();

builder
    .ForAllCommands()
    .UseAllAvailableBuses()
    .HandleErrorWith(e => { Console.WriteLine(e); })
    .IsSecurityCritical()
    .SerializeWith<JsonDispatcherSerializer>();

    //Get the configuration
var config = builder.Build();

    //Apply it to current system
    new Bootstrapper()
            .ConfigureDispatcher(config)
            .Bootstrapp();
```

On récupère la configuration en appelant la méthode `Build()`. Il est possible de spécifier un paramètre "strict" au build de la configuration. La définition de sa valeur à "true" vérifie que tous les events et les commands sont assignés à un bus minimum. Ca permet d'assurer qu'il n'y a pas de type qui sont orphelins et ne seront pas traités lors d'un dispatch. A noter que cette valeur est mise à false par défaut.

Une fois la configuration récupérée, on la passe en paramètre au bootstrapper pour l'appliquer au système (voir l'article sur le *Bootstrapper* pour plus de détails).

1.10 Bootstrapper

1.10.1 Généralités

Le bootstrapper est le point d'entrée principal du système, qui permet de le configurer comme voulu afin de s'adapter à votre application, votre infrastructure et votre métier. Il permet de mettre en relation les différentes extensions de CQELight avec le coeur du système. Il existe plusieurs types d'extensions :

- **Container IoC** : il s'agit des extensions qui résident au coeur de tout le système, car c'est dans cette extension que sont enregistrés les liens entre les abstractions et les implémentations.
- **Bus** : il s'agit des extensions qui définissent un bus de messaging, utilisé pour transporter les évènements et les commandes dans le système.
- **DAL** : il s'agit des extensions qui permettent d'abstraire l'accès aux donnés, utilisés majoritairement par la couche Query, et de fournir des implémentations pour la couche repository.
- **EventStore** : il s'agit des extensions qui managent le système évènementiel, se chargeant de la persistance et de la récupération des évènements qui sont arrivés dans le système.
- **Autre** : il s'agit de toutes les extensions voulant profiter de ce qu'on CQELight en terme de flexibilité et d'outils pour permettre une intégration facilitées. On y trouvera également des extensions commerciales qui ne répondent pas à un type ci-dessus.

Toutes ces extensions doivent être configurées et injectées dans le boostrapper, par le biais d'une classe implémentant l'interface IBoostrapperService. Cette implémentation doit définir le type d'extension dont il s'agit ainsi qu'une méthode callback qui effectue de façon lazy le boostrapping (ceci étant dû à des mécanismes internes d'initialisation). Cette méthode d'initialisation prends mets à disposition un paramaètre, de type BootstrappingContext que les extensions peuvent exploiter pour avoir plus d'informations au moment de leur bootstrapping propre.

Note : A noter qu'aucune extension n'est obligatoire. Cependant, le système sera limité voire inopérant s'il manque des services. Le cas de l'IoC doit être traité avec une extrêmement vigilance car ce n'est pas toujours disponible. L'information est à votre disposition lors de la méthode de bootstrapping, dans le BootstrappingContext.

Voici une définition de bootstrapper « classique »

```
new Bootstrapper()
   .ConfigureCoreDispatcher(GetCoreDispatcherConfiguration())
   .UseInMemoryEventBus(GetInMemoryEventBusConfiguration())
   .UseInMemoryCommandBus()
   .UseEFCoreAsMainRepository(new AppDbContext())
   .UseSQLServerWithEFCoreAsEventStore(Consts.CONST_EVENT_DB_CONNECTION_STRING)
   .UseAutofacAsIoC(c => { })
   .Bootstrapp();
```

La méthode Bootstrapp retourne une liste de notifications. Cette liste contient un ensemble de notifications émises soit par le système soit par les extensions. Les notifications sont de trois niveaux : `Info`, `Warning` et `Error`.

Note : Il reste dans la responsabilité du développeur de consulter et d'exploiter cette liste, aucune exception n'était renvoyée lors du process de bootstrapping. Il est recommandé d'arrêter le processus de démarrage d'une application si une notification de type "Error" survient.

Il y a en également plusieurs paramètres possibles pour initialiser le bootstrapper :

- On peut lui affecter une valeur pour "strict". Le passage de ce paramètre à true impliquera que, hors extensions de type Bus et Autre, il est impossible d'enregistrer plus d'une extension pour un type de service donné (par exemple, impossible d'avoir deux extensions de type IoC).

— On peut lui affecter une valeur pour “optimal”. Le passage de ce paramètre à true impliquera que, lors de la méthode Bootstrapp, il y aura vérification qu’au moins un service de chaque type sera enregistré (exception faite du type « Autre »). Le système fonctionnera donc de la meilleure façon possible.

1.11 Créer vos propres extensions

1.11.1 Généralités

CQELight a été initialement conçu afin d’être hautement extensible, en fonction des besoins de chaque projet et de chaque équipe. A cet effet, il est possible, tout comme les extensions officielles, de créer vos propres extensions. Le processus se veut être assez simple.

Pour développer une extension, il est nécessaire de savoir de quel type d’extension il s’agit. Il y a cinq types d’extensions possibles :

— Gestionnaire d’IoC
— Service de bus messaging
— Service d’accès aux données
— Event store
— Autre

Une fois le type d’extension défini, il faut passer par plusieurs étapes intermédiaires afin d’en créer une. Afin de conserver la logique modulaire, il est fortement conseillé de faire un package par extension, au cas où les besoins de votre projet viendrait à évoluer. Une extension est un nouveau projet de type “Bibliothèque de classes” (de préférence en .NET Standard 2.0). Une fois le nouveau projet créé dans Visual Studio, il faut y définir les élements nécessaire pour la configuration :

— Une classe service qui sera ajoutée à la collection du bootstrapper
— La méthode d’extension du Boostrapper qui vous permettra de configurer votre extension de façon fluide et moderne (du style UseXXX)

1.11.2 Création de l’extension

Création de la classe de service

Comme expliqué dans la doc sur le bootstrapper, chaque extension possède une classe de service qui implémente `IBoostrapperService`, dans le but d’apporter une certaine cohésion dans la gestion de votre extension. Cette classe doit implémenter l’interface `CQELight.IBootstrapperService`

```
internal class MyAwesomeBusExtensionService : CQELight.IBootstrapperService
{
    public BootstrapperServiceType ServiceType => BootstrapperServiceType.Bus;

    public Action BootstrappAction { get; internal set; } = (ctx) => { };
}
```

L’interface impose la défintion de deux membres :

— `ServiceType`, correspondant à l’énumération pour préciser de quel type de service il s’agit.
— `BootstrappAction`, étant l’action éxécutée lors du bootstrapping. Cette méthode possède à sa disposition le `BootstrappingContext` permettant d’avoir plus d’infos sur l’état du système lors de votre bootstrapp.

Lorsque cette classe de service est faite, il est nécessaire d’ajouter une instance de cette dernière dans la collection des services du bootstrapper (dans la méthode d’extension de configuration).

Note : Il est obligatoire de passer par une méthode de bootstrapping qui sera exécutée plus tard dans le process, afin de permettre au système de faire des évaluations et traitements avant que chaque extension soit réellement initialisée.

Si vous prenez le parti de faire directement des instanciations lors de votre méthode d'extension, vous vous exposez à des effets de bord indésirables.

Méthode d'extension du bootstrapper

Toutes les extensions étant bootstrappées au lancement de l'application selon un ordre défini par le framework, il faut fournir un point d'entrée pour indiquer que l'on veut utiliser la vôtre. Il est recommandé de procéder au bootstrapping au lancement de votre application afin de permettre que la configuration soit centralisée, et d'avoir le système prêt le plus tôt possible (dans le `Startup` d'une application AspNet ou dans le `App.xaml` d'une application WPF par exemple). Afin de configuer le bootstrapper, il faut appeler les méthodes nécessaires sur une instance de la classe `Bootstrapper`, qui sont généralement des méthodes d'extensions. Il vous faut alors faire une méthode d'extension sur la classe `Bootstrapper` pour permettre d'appeler l'initialisation de votre extension.

Cette méthode d'extension s'applique sur la classe `CQELight.Bootstrapper`, et doit retourner l'instance initiale, afin de permettre d'enchainer les appels de configuration. Le but ici est fourni la méthode de callback qui sera appelée par le système dans l'ordre défini (qui n'est pas l'ordre d'appel des méthodes d'extension) pour préparer le contexte général propice à votre extension (injection de type dans le container IoC, définition de variable statiques, etc...). Vous devrez alors utiliser ajouter votre classe de service au bootstrapper après l'avoir implémentée.

Note : Attention : l'utilisation d'un container IoC n'est pas obligatoire pour utiliser CQELight, il s'agit d'une extension au même titre qu'une autre. De ce fait, il est fortement recommandé que, même si vous utilisez l'injection de dépendances dans votre extension, vous n'en fassiez pas quelque chose obligatoire (sauf si vous avez la maitrise totale sur le système globale), de peine de se priver d'un public potentiel pour votre extension. Vous pouvez consulter le BoostrappingContext pour savoir si une extension IoC est définie.

```
public static Bootstrapper UseMyAwesomeExtension(this Bootstrapper bootstrapper, ...
↪custom params...)
{
    var service = new MyAwesomeExtensionService();
    service.BootstrappAction += (ctx) =>
       {
           bootstrapper.AddIoCRegistration(new␣
↪TypeRegistration(typeof(MyImplementation), typeof(IMyAbstraction),␣
↪typeof(MyImplementation)));
       };

    if (!bootstrapper.RegisteredServices.Any(s => s == service))
    {
        bootstrapper.AddService(service);
    }
    return bootstrapper;
}
```

Par convention, cette classe se trouve à la racine de votre projet et se nomme `Bootstrapper.ext`. Il faut cependant préciser que cette classe ne contient que vos méthodes d'extensions et que le nommage ne change rien au fonctionnement général.

Note : Il est recommandé de faire une méthode d'extension sur le bootstrapper et de retourner l'instance en paramètre pour permettre une fluent configuration. Cependant, rien ne l'oblige dans votre propre projet. C'est par contre un élement obligatoire si vous souhaitez que votre extension rejoigne la liste officielle des extensions CQELight.

Définition du contenu de l'extension

Ici, il s'agit de votre extension, c'est à vous d'en définir son implémentation.

Par contre, vous pouvez voir plusieurs exemples sur comment réaliser une extension sur notre GitHub (chaque package est une extension).

Après demande de votre part (remplir une issue sur notre GitHub), vous pouvez demander à ce que votre extension rejoigne la collection officielle des extensions CQELight, publiée sur NuGet, avec la documentation hébergée par Hybrid Technologies Solutions. Ceci passe par une étape de review de code et de test, ainsi que de la mise en confirmité avec nos standards. Vous pouvez également participer à l'élaboration des extensions officielles existantes qui sont open-source.

Lors de la création de votre méthode bootstrapping, vous aurez accès à un contexte de bootstrapping. Ce contexte contient un ensemble d'information vous permettant de configurer plus finement votre extension. Vous y trouverez entre autre :

— Les flags passés au constructeur du bootstrapper, strict et optimal (voir documentation du bootstrapper pour comprendre la signification). Ces flags vous permettent de configurer votre extension en fonction des contraintes voulues par l'appelant général.
— Une méthode `IsServiceRegistered` qui permet de savoir si un service d'un type donné a déjà été défini (comme par exemple un service de type IoC pour effectuer des injections IoC).
— Une méthode `IsAbstractionRegisteredInIoC` qui permet de savoir si un type abstrait a déjà été défini dans le container du bootstrapper. Attention cependant, cette méthode ne garantit en rien qu'un telle association n'ait pas été faite en dehors du bootstrapper. Le cas échéant, l'information n'est pas disponible par le biais de cette extension.

Selon les flags qui vous sont passés et les besoins de votre extension, il est possible d'ajouter des notifications au niveau du bootstrapper. La classe `CQELight.Bootstrapper` expose deux méthodes, `AddNotification` et `AddNotifications` qui vous permettent de réaliser cette opération. Vous pouvez créer une notification en précisant le type de notification (Info, Warning, Error) ainsi qu'un message, et il également possible de fourni le type de service qui a créé cette notification.

1.11.3 Spécificités de chaque type

Extension IoC

Si vous développez une extension pour la gestion d'un container IoC, il est impératif de gérer les types qui ont été enregistrés dans le bootstrapper par les autres extensions. Voir la documentation sur l'*Inversion of Control* pour savoir les différents types d'enregistrements à gérer.

Une extension de type IoC doit également prendre en charge les interfaces `IAutoRegisterType` et `IAutoRegisterTypeSingleInstance`, qui sont des raccourcis pour permettre l'enregistrement de type dans le container IoC sans en maitriser la particularité.

1.12 Tests unitaires et fonctionnels

Chez Hybrid Technologies Solutions, nous nous efforçons de développer avec des tests unitaires et fonctionnels. A cet effet, il nous était impensable d'imaginer que si d'un côté, nous réalisons un framework pour aider les développeurs à se focaliser sur le métier, de l'autre nous ne prévoyons pas la possibilité de facilement tester le code écrit avec notre framework.

Pour répondre à ce besoin, nous avons créé un package à ajouter à vos projets de tests unitaires : **CQELight.TestFramework**. Ce package contient un certain nombre d'outils que nous avons jugé utile de rajouter pour vous aider à faire vos tests unitaires.

..note : : A la différence de l'ensemble du framework, nous avons ajouté certains packages que nous utilisons quotidiennement en test unitaire, à savoir Moq et FluentAssertions. Nous n'avons pas trouvé cela problématique considérant la popularité de ces packages. Cependant, si plus tard, cela poserait problème, nous créerons des « sous-packages » pour ces points précis. De fait, en installant notre TestFramework, vous installerez automatiquement ces deux packages également, et vous profiterez de certaines méthodes d'extension que nous avons réalisées sur ces derniers.

1.12.1 BaseUnitTestClass

La première chose que nous mettons à votre disposition est une classe de base pour vos tests unitaires, `CQELight.TestFramework.BaseUnitTestClass`. Cette classe effectue quelques actions automatiquement à la construction.

La classe `CQELight.TestFramework.UnitTestTools` vous fournit deux flags qui permettent de détecter le mode de fonctionnement de votre classe de test. Ces flags peuvent être utile à n'importe quelle partie pour savoir le contexte.

— Le flag `IsInUnitTestMode` est à vrai dès lors qu'une instance a été construite.
— Le flag `IsInIntegrationTestMode` se mets à vrai si le nom de votre projet contient .Integration.

Ces flags sont automatiquement déterminé dans la construction de la classe de base `BaseUnitTestClass`.

Note : Il s'agit ici d'une convention que nous avons adoptée, permettant de distinguer les tests unitaires des tests d'intégration en se basant sur le nom du projet. Cela peut s'avérer utile pour vos tests nécessitant un contexte particulier (accès au système de fichier, base de données, connexion réseau, ...) et qui ne peuvent pas être exécutés n'importe où.

Afin de vous permettre d'utiliser l'IoC facilement, nous avons développé une couche factice que vous pouvez alimenter selon les besoins de vos tests. Le constructeur de la classe `BaseUnitTestClass` créé une instance de notre scope factory de test et initialise le système d'IoC avec celle-ci (hors tests d'intégration). Si vous désirez que ce comportement ne soit pas exécuté, vous pouvez préciser le paramètre constructeur, `disableIoC`, à true pour empêche cette initialisation. Une fois celle-ci faite, vous aurez accès au membre protégé `_testFactory` dans lequel vous pouvez ajouter les enregistrements que vous avez besoin pour vos tests (en ajoutant une ou plusieurs valeurs dans la propriété Instances qui alimenteront automatiquement vos scope).

Exemple

```
public class MyUnitTests : BaseUnitTestClass
{
    public MyUnitTests()
    {
        var myAbstractionMock = new Mock<IMyAbstraction>();
        _testScopeFactory.Instances.Add(typeof(IMyAbstraction), myAbstractionMock.
↪Object);
    }
}
```

Note : Il s'agit ici d'une implémentation factice destinée à simplifier les tests unitaires et elle ne saurait en aucun cas se substituer à un vrai système d'IoC ni prétendre en avoir les mêmes possibilités (injection automatique d'abstractions dans le constructeur, injection dans les propriétés, ...). Pensez à conserver la vision test unitaire pour que cela soit adapté. Si vous besoins sont plus complexe, il s'agit probablement d'un test d'intégration, et dans ce cas précis, il est recommandé d'utiliser un vrai container IoC.

Finalement, il est possible d'ajouter un enregistrement dans ce scope factice en dehors du constructeur, à l'aide de la méthode protégée `AddRegistrationFor`. Cela vous permet d'ajouter une implémentation dans votre test directement juste avant l'exécution

```
[Fact]
public void MyUnitTest()
{
    var myAbstractionMock = new Mock();
    AddRegistrationFor(myAbstractionMock.Object);
}
```

La méthode `CleanRegistrationInDispatcher` est un raccourci qui permet de vider le container statique du `CoreDispatcher` qui contient les instances enregistrées directement dedans (de type `IMessageHandler`, `IDomainEventHandler` et `ICommandHandler`).

1.12.2 IoC

Comme nous l'avons vu précédemment, nous mettons à disposition un `TestScope` et un `TestScopeFactory`. Ces deux classes vous permettront de simuler le comportement du container IoC au niveau méthode. Le `TestScope` prends en paramètre de constructeur un dictionnaire de concordance entre un type et une instance, vous permettant de retourner l'implémentation désirée pour le test selon un type donné. Le `TestScopeFactory` permet d'avoir cet enregistrement à un niveau plus général et injectera ce dictionnaire de concordance à chaque scope de test créé

```
var testScopeFactory = new TestScopeFactory();
testScopeFactory.Instances.Add(typeof(IMyAbstraction), myImplementationVar);

var scope = testScopeFactore.CreateScope(); // Scope will contain myImplementationVar
↪"registration"

var instance = scope.Resolve(); // Instance will be same object as myImplementationVar
var instance2 = new TestScope(new Dictionary{ {typeof(IMyAbstraction),␣
↪myImplementationVar} }); // will be the same result as previous line
```

Une instance du `TestScopeFactory` est mis à disposition dans le BaseUnitTestClass.

1.12.3 Bus

Il peut arriver que vous ayez besoin, dans le cadre bien définit d'une méthode donnée, directement d'un bus. Pour répondre à ce problème, nous avons fourni deux implémentation test, `FakeCommandBus` et `FakeEventBus`. Ces deux bus implémente respectivement `ICommandBus` et `IEventBus`, et fournissent sous forme d'un `IEnumerable` public la liste des commandes/events qui ont publiées par leur biais.

Note : Si vous avez des besoins de tests plus avancés, nous vous recommandons d'utiliser les bus in-memory, plus complexes et plus lents, mais plus extensibles et configurables. L'utilisation des FakexxxBus est recommandé uniquement pour des tests unitaires extrêmement simples où le bus est directement passé en tant que dépendance à la classe.

1.12.4 Test du dispatch

Bien que nous ayons vu ci-dessus l'existence de deux bus de tests pour simuler les envois d'informations dans le système, il peut-être utile de se placer un cran dessus et vérifier le comportement du dispatcher. Il y a deux modes de fonctionnement pour le dispatcher : l'utilisation d'une instance qui implémente `IDispatcher` et l'utilisation de la version statique `CoreDispatcher`.

Ce qui va intéresser le développeur est de savoir si sa commande/son évenement a bien été publié, si plusieurs commandes/évenements ont été publiés ou au contraire, si aucun ne l'ont été, afin de s'assurer du comportement attendus. Nous avons créé une classe statique, `Test`, qui permet de s'assurer de cela.

La classe test s'applique sur un contexte d'exécution donné (méthodes `When` et `WhenAsync`). Il est possible de passer un mock d'une instance de `IDispatcher` afin d'effectuer les vérifications sur ce dernier plutôt que sur le `CoreDispatcher` statique. Lorsque le contexte est créé, on récupère la possibilité d'effectuer un test sur l'exécution du contexte. Toutes les méthodes disposent de la possibilité de passer un timeout en millisecondes afin d'éviter d'avoir des tests trop longs (fixé par défaut à 1 sec). La liste des méthodes de test possibles sont :

- `ThenNoEventShouldBeRaised` : vérifie qu'aucun évènement n'est levé à la suite de l'appel du contexte
- `ThenNoCommandAreDispatched` : vérifie qu'aucune commande n'est envoyée à la suite de l'appel du contexte
- `ThenEventShouldBeRaised<T>` : vérifie qu'un évènement, du type donné, est levé à la suite de l'appel du contexte. Si plusieurs évènements sont publiés, uniquement le dernier évènement de type T sera renvoyé
- `ThenCommandIsDispatched<T>` : vérifie qu'une commande, du type donné, est publiée à la suite de l'appel du context. Si plusieurs commandes sont publiées, uniquement la dernière de type T sera renvoyée
- `ThenEventsShouldBeRaised` : vérifie que que plusieurs évènements sont publiés à la suite de l'appel du contexte
- `ThenCommandsAreDispatched` : vérifie que plusieurs commandes sont publiées à la suite de l'appel du contexte
- `ThenNoMessageShouldBeRaised` : vérifie qu'aucun message n'a été envoyé à la suite de l'appel du contexte. Attention, cette méthode n'est évaluée que sur le CoreDispatcher
- `ThenMessagesShouldBeRaised` : vérifie que plusieurs messages ont été envoyés à la suite de l'appel du contexte. Attention, cette méthode n'est évaluée que sur le CoreDispatcher
- `ThenMessageShouldBeRaised<T>` : vérifie qu'un mesage, du type donné, est envoyée à la suite de l'appel du contexte. Attention, cette méthode n'est évaluée que sur le CoreDispatcher

```
var evt = await Test.WhenAsync(myAsyncMethod).ThenEventShouldBeRaised();
//Perform tests on evt instance ...
```

1.12.5 Méthode d'extensions

Pour conclure, nous avons ajouté un ensemble de méthodes d'extensions. Ces méthodes concernent aussi bien nos plugins que nos assemblies de base que les packages communautaires que nous incluons par défaut.

Au niveau DDD, la méthode `ClearDomainEvents` permet, sur une instance d'un `AggregateRoot`, de nettoyer la collection d'évènements ajoutés par le biais des méthodes AddDomainEvent de l'aggregat, vous permettant de vider la collection d'évènements entre plusieurs appels pour faire vos assertions.

Au niveau DAL, nous avons donné la possibilité de

- `FakePersistenceId` : permet d'effectuer le set de la propriété Id d'un PersistableEntity
- `SetupSimpleGetReturns` : permet de définir extrêmement facilement sur un mock d'un `IDataReaderRepository` ce que la méthode `GetAsync` doit renvoyer (en fournissant une liste d'élément finie in-memory)
- `VerifyGetAsyncCalled` : à l'instar de la méthode précédente, permet de vérifier extrêmement facilement si la méthode `GetAsync` a été appelée sur un mock d'un `IDataReaderRepository`

Au niveau MVVM (package `CQELight.TestFramework.MVVM`), nous avons définit une méthode, `GetStandardViewMock` qui permet de retourner un mock par défaut de l'interface `IView`. La spécificité de ce mock est qu'il contient déjà la méthode de callback pour la méthode `PerformOnUIThread`, de façon à ce que cette dernière s'exécute de façon systématique en contexte de test unitaire.

1.13 Scénario complet

Afin d'explorer au maximum CQELight de façon assez simple, nous allons faire, dans un contexte mono-applicatif desktop console, un programme de test qui démontrera les concepts que nous avons vu précédemment.

L'idée va être de développer une petite application permettant la gestion d'un arbre généalogique, de façon ultra simplifiée. On se contentera uniquement de lister les personnes d'une famille, avec ses infos de naissance et leur date de décès.

Au niveau des informations de naissance, on se contente de stocker date et lieu de naissance. La date de décès peut ne pas être renseignée. Il est bien entendu impossible d'avoir une date de décès inférieure à la date de naissance. On considère à cet effet que deux personnes sont nées « de la même façon » si elles sont nées au même endroit le même jour.

La famille est identifiée de façon unique par le nom. Il ne peut pas y avoir deux familles avec le même nom. Au niveau des personnes, en plus des infos de naissance, on stockera uniquement le prénom. Il est possible d'avoir plusieurs personnes avec le même prénom dans la même famille, si les informations de naissance sont différentes, sinon, c'est qu'il s'agit d'un doublon, et ce n'est pas autorisé.

Dans les pages suivantes, on va explorer ce sujet petit à petit pour le modéliser et utiliser CQELight pour arriver à nos fins. Nous allons créer un système event-sourcé, séparé en CQRS et hautement extensible.

Vous trouverez l'ensemble du code sur notre repository GitHub.

1.14 Modélisation du domaine

Avant toute chose, il convient de modéliser le domain métier par les objets le représentant. Cette modélisation est volontairement simple et couvre explicitement les trois objets de base (Aggregate, Entity et ValueObject).

1.14.1 ValueObject

Afin de pouvoir stocker des valeurs et effectuer des traitements, nous allons découper notre domaine dans ces trois types d'objet. Pour rappel, un ValueObject (VO dans la suite du texte) est identifié par l'unicité métier de l'ensemble de ses membres et doit être immuable. La notion qui colle le plus à une VO dans notre exemple est la gestion des informations de naissance (date et lieu), étant donné que le scénario considère que deux personnes sont nées de la même façon si elles sont nées le même jour au même endroit.

Le VO qui en découle est le suivant

```
using CQELight.Abstractions.DDD;
using System;

namespace Geneao
{
    public class InfosNaissance : ValueObject<InfosNaissance>
    {

        public string Lieu { get; private set; }
        public DateTime DateNaissance { get; private set; }

        public InfosNaissance(string lieu, DateTime dateNaissance)
        {
            if (string.IsNullOrWhiteSpace(lieu))
            {
```

(suite sur la page suivante)

(suite de la page précédente)

```
                throw new ArgumentException("InfosNaissance.Ctor() : Lieu requis.",␣
↪nameof(lieu));
            }

            Lieu = lieu;
            DateNaissance = dateNaissance;
        }

        protected override bool EqualsCore(InfosNaissance other)
        => other.DateNaissance == DateNaissance && other.Lieu == Lieu;

        protected override int GetHashCodeCore()
        => (typeof(InfosNaissance).AssemblyQualifiedName + DateNaissance + Lieu).
↪GetHashCode();
    }
}
```

Comme on peut le constater, un VO doit hériter de la classe `CQELight.Abstractions.DDD.ValueObject` pour avoir le comportement de base. Deux méthodes sont à overrider : `EqualsCore` et `GetHashCodeCore`, afin de permettre de redéfinir niveau métier ce qui fait foi pour cet objet. Afin de garantir son immuabilité, vous remarquerez que les setter sont privés et uniquement le constructeur peut définir ces valeurs. C'est voulu, c'est afin qu'il n'y ait aucun changement au cours de la vie d'un objet. De façon générale, lors de la création d'un VO et son stockage auprès d'une entité, on s'assure effectivement que la même valeur soit véhiculée. Si nous avons besoin de changer une information, on créera une nouvelle instance.

1.14.2 Du bon choix des types

Dans notre petit domaine, nous allons utiliser un Guid pour définir une personne et un string pour identifier une famille. Il est nécessaire de ne pas utiliser les types primitifs car cela n'a aucun sens métier fort. Au lieu de ça, il est préférable de les encapsuler pour leur donner une vraie définition

```
using System;

namespace Geneao.Identity
{
    public struct PersonneId
    {
        public Guid Value { get; private set; }

        public PersonneId(Guid value)
        {
            if (value == Guid.Empty)
                throw new InvalidOperationException("PersonneId.ctor() : Un␣
↪identifiant valide doit être fourni.");
            Value = value;
        }

        public static PersonneId Generate()
            => new PersonneId(Guid.NewGuid());
    }
}
```

```
using System;
```

(suite sur la page suivante)

(suite de la page précédente)

```
namespace Geneao.Identity
{
    public struct NomFamille
    {
        public string Value { get; private set; }

        public NomFamille(string value)
        {
            if (string.IsNullOrWhiteSpace(value) || value.Length > 128)
                throw new InvalidOperationException("NomFamille.ctor() : Un nom de famille correct doit être fourni (non vide et inférieur à 128 caractères).");
            Value = value;
        }
        public static implicit operator NomFamille(string nom)
            => new NomFamille(nom);
    }
}
```

Ces deux classes nous permettent d'utiliser, dans l'ensemble de notre modèle, un identifiant unique pour une Personne et pour une Famille. L'intérêt d'avoir encapsulé les types simples dans ces classes d'identité nous permet de respecter le principe Single Responsability (pour la vérification de la donnée), et nous évitera ultérieurement des erreurs de code (comme par exemple, dans un méthode, avec un string pour le nom et un string pour le prénom, il est très facile d'intervertir les paramètres, alors que si l'un de ceux-ci est un `NomFamille`, on évite ce désagrément).

Note : Dans le code précédent, nous avons rajouté un opérateur implicite de conversion entre un string et la classe NomFamille. Le but de cette opération est de faciliter le développement en faisant `NomFamille nom="famille1"`. Cependant, l'effet négatif de ce changement peut être l'inversion de paramètre dont nous parlions précédemment.

1.14.3 Entity

Pour rappel, une entité véhicule des données (muable) et un comportement. Une entité n'est pas censée exister en dehors d'un agrégat donné. Dans notre gestion de famille, la notion de personne s'y porte le plus, car dans ce contexte uniquement, une personne n'est pas censée exister en dehors d'une famille.

Note : Attention, il est toujours nécessaire, quand on modélise le domaine, de rester concentré sur le contexte qu'on est entrain d'étudier et ne pas penser au système en général. Plusieurs entités autour d'un même objet réel vont être modélisées différement dans les différents contextes. Si une ou plusieurs entités seraient totalement identiques d'un contexte à l'autre, on peut alors parler de Shared Kernel, c'est à dire d'informations communes partagées car véhiculant un sens métier « universel » dans notre système.

```
using CQELight.Abstractions.DDD;
using System;

namespace Geneao
{
    public class Personne : Entity<PersonneId>
    {
        public string Prenom { get; internal set; }
        public InfosNaissance InfosNaissance { get; internal set; }
```

(suite sur la page suivante)

(suite de la page précédente)

```
        internal Personne() { }

        public static Result DeclarerNaissance(string prenom, InfosNaissance␣
↪infosNaissance)
        {
            if (string.IsNullOrWhiteSpace(prenom))
            {
                return Result.Fail(DeclarationNaissanceImpossibleCar.AbsenceDePrenom);
            }

            if (infosNaissance == null)
            {
                return Result.Fail(DeclarationNaissanceImpossibleCar.
↪AbsenceInformationNaissance);
            }

            return Result.Ok(new Personne(PersonneId.Generate())
            {
                Prenom = prenom,
                InfosNaissance = infosNaissance
            });
        }
    }
}
```

Note : Dans le bloc précédent, nous avons fait le choix d'utiliser la structure Result fournie avec CQELight au lieu des exceptions, et ce afin d'éviter de faire rentrer la mécanique de gestion des exceptions, qui peut être lourde en terme de performances. De plus, `DeclarerNaissance` est une fonction métier, elle a donc du sens à retourner un résultat métier plutôt qu'un résultat technique.

Le code est assez explicite pour décrire le comportement de cette entité. Ici, on considère la clé comme étant un `PersonneId`. Ceci est fait en héritant de la classe `CQELight.Abstractions.DDD.Entity` avec l'id désiré. Ensuite, on fait en sorte que les données soit visibles de l'extérieur, mais modifiable uniquement de l'intérieur de l'assembly (pour que l'agrégat puisse les modifier si nécessaire). Finalement, on rends le constructeur internal (encore une fois pour les besoins éventuels de l'agrégat) et on fait une factory qui a un sens métier fort, avec les contrôles associés.

Note : Ici, on pourrait faire un raccourci rapide et considérer que c'est la responsabilité de l'agrégat de s'assurer que la création d'une personne est validée par lui seul. Nous préférons découper notre domaine de telle façon que chaque classe gère le contrôle de ses données propres, avec la possibilité pour l'agrégat parent d'en faire des modifications si nécessaire.

1.14.4 Aggregate

Pour rappel, la notion d'AggregateRoot est la partie publique de notre contexte courant, elle doit représenter une frange du métier de ce contexte. C'est donc l'objet qui sera désigné comme `AggregateRoot` qui exposera publiquement les moyens d'entrer en contact avec le domaine (modification ou lecture). De la même façon, un agrégat étant un regroupement métier, il est essentiel qu'il soit le garant de son état interne.

Nous allons gérer des personnes regroupées en famille plutôt que des personnes de façon individuelle. Une famille,

dans ce contexte, regroupe un ensemble de personne, c'est donc notre AggregateRoot

```
using CQELight.Abstractions;
using CQELight.Abstractions.DDD;
using System;
using System.Collections.Generic;
using System.Linq;

namespace Geneao
{
    public class Famille : AggregateRoot<NomFamille>
    {
        public IEnumerable Personnes => _state.Personnes.AsEnumerable();

        private FamilleState _state;

        private class FamilleState : AggregateState
        {

            public List Personnes { get; set; }

            public FamilleState()
            {
                Personnes = new List();
            }
         }

         public Famille(NomFamille nomFamille, IEnumerable personnes = null)
         {
             Id = nomFamille;
             _state = new FamilleState
             {
                 Personnes = (personnes ?? Enumerable.Empty()).ToList()
             };
         }

         public static Result CreerFamille(string nom, IEnumerable personnes = null)
         {
             return Result.Ok(new Famille(new NomFamille(nom), personnes));
         }

         public Result AjouterPersonne(string prenom, InfosNaissance infosNaissance)
         {
             if(!_state.Personnes.Any(p => p.Prenom == prenom && p.InfosNaissance ==␣
↪infosNaissance))
             {
                _state.Personnes.Add(Personne.DeclarerNaissance(prenom,␣
↪infosNaissance));
             }
             return Result.Ok()
         }
    }
}
```

Notre agrégat est censé gérer la cohérence d'une famille, dans ce domaine. De fait, il est nécessaire de vérifier que la personne qu'on tente d'ajouter n'existe pas déjà dans cette famille. La factory de création permet, depuis un nom et une liste de personne, de récupérer un agrégat domaine de `Famille`. A noter ici que si le nom est incorrect, la vérification est faite par la partie domaine de l'identité et renvoie l'exception directement, sans traitement. Ce comportement sera

géré correctement lors de la mise en place des évènements domaine.

Note : C'est normalement ici qu'on fait la gestion des évènements. Nous prendrons cet exemple pour en parler ultérieurement dans la partie de la documentation sur les events & les commands.

Avec cette mise en place initiale, notre modèle est constitué et on peut continuer à l'enrichir selon l'évolution du métier. Dans notre cas, par exemple, et à titre d'exercice, on peut rajouter la notion de mariage, la notion d'enfant/parent, etc... Libre à vous de continuer sur cette lancée et de continuer cet exercice !

Note : Important : il faut garder en tête qu'une modélisation est toujours imparfaite. De ce fait, nous serons amenés tout au long de cet exercice à retoucher ce code. Il ne faut s'y « attacher » au point de vouloir le laisser tel quel. D'autre part, nous n'avons créé aucun test unitaire, c'est un bon exercice de créer des tests pour vérifier le code existant et s'assurer que nos prochains changements ne casseront pas le domaine.

1.15 Commands

Notre domaine étant modélisé, il est actuellement nécessaire de faire un appel direct, avec un couplage fort, à l'agrégat Famille pour récupérer une Famille (méthode `CreerFamille` ou constructeur). Afin d'éviter ce couplage fort avec notre domaine et permettre au domaine d'évoluer sans impacter les appelants, nous allons définir une commande. Pour rappel, une commande est un simple DTO chargé de véhiculer les informations suffisantes pour s'exécuter et elle doit également vérifier l'intégrité des données qu'elle transporte (pas de vérification métier à son niveau, juste de la vérification technique).

Notre première commande porterait donc la notion de création d'une famille

```
using CQELight.Abstractions.CQS.Interfaces;
using System;

namespace Geneao.Commands
{
    public sealed class CreerFamilleCommand : ICommand
    {
        public string Nom { get; private set; }

        private CreerFamilleCommand() { }

        public CreerFamilleCommand(string nom)
        {
            if (string.IsNullOrWhiteSpace(nom))
            {
                throw new ArgumentException("CreerFamilleCommand.ctor() : Un nom doit␣
↪être fourni.", nameof(nom));
            }
            Nom = nom;
        }
    }
}
```

Plusieurs élements sont importants dans cette portion de code :

— Tout d'abord, les properties sont uniquement affectées depuis le constructeur, ce qui permet d'assurer leur validité. Il est essentiel de faire cette vérification, car on considère que dès lors qu'une commande arrive dans un handler, elle est valide (c'est à dire que le handler n'a pas à vérifier le contenu des membres ni leur

cohérence.) Attention, ce n'est pas à la commande, simple DTO, de faire des vérifications métier (comme ici, par exemple, la taille du nom de famille).
— Il y a un constructeur, sans paramètres, privé. En effet, il n'est pas improbable que votre commande passe par des passerelles de communication (bus, appel API, ...), et devra donc être sérialisée/déserialisée. Il faut donc un point d'entrée pour le moteur de sérialisation, et les setters inaccessibles pour une utilise normale mais utilisable par le moteur.
— La classe implémente l'interface `CQELight.Abstractions.CQS.Interfaces.ICommand`, qui ne contient rien du tout. Elle est uniquement là pour un typage fort. Attention lors de l'import de l'using de ne pas prendre celui de `System.Windows.Input`.
— Le nom de la classe d'une commande utiliser toujours un verbe à l'infinitif, car on « ordonne » au système de faire une action. Grâce à ce nommage, on peut se passer du suffixe *Command*, selon les affinités des équipes, ou les normes de code. Dans l'exemple nous gardons le suffixe pour que vous puissiez facilement retrouver vos objets dans le projet.
— Etant donné qu'une commande est l'entrée dans le sytème, il faut verrouiller les accès non autorisés avec des données farfelues, du coup, on définie la classe comme sealed, pour éviter des héritages inattendus depuis l'extérieur du système.

On peut dès lors propager cette commande dans le système une fois qu'elle est créée. Ceci se fait très simplement

```
await CoreDispatcher.DispatchCommandAsync(new CreerFamilleCommand("NomTest")).
↪ConfigureAwait(false);
```

Note : Dans cet exemple, nous utilisons l'API statique du dispatcher, qui est effectivement plus simple d'accès (pas d'instanciation). Il est recommandé, dans la grande majorité des cas, d'utiliser l'API d'instance (avec un `IDispatcher` injecté en paramètre de constructeur, ou en utilisant directement la classe `BaseDispatcher` ou toute autre implémentation que vous en auriez faite) afin d'éviter les accès concurrentiels et les problèmes associés (lock, performances, ...)

De cette façon, la commande de demande de création d'une personne avec les informations renseignées va être propagée dans le système. Par contre, il n'y a aucun point d'arrivée pour traiter cette commande. Ici, il y a deux possibilités : soit notre Aggregate peut réagir directement à la commande (en implémentant l'interface `ICommandHandler`), soit on crée un handler spécifique pour traiter cette commande. La création du handler est fortement recommandée pour la quasi totalité des cas, car passer par cette étape intermédiaire permet de résoudre les problèmes infrastructuraux associés au traitement de la commande (chargement d'informations depuis une source de données, récupération d'un aggregat complexe, ...)

```
using CQELight.Abstractions.CQS.Interfaces;
using Geneao.Commands;
using Geneao.Domain;
using System.Threading.Tasks;

namespace Geneao.Handlers.Commands
{
    public class CreerFamilleCommandHandler : ICommandHandler<CreerFamilleCommand>
    {
        private static List<Famille> _familles = new List<Famille>();
        public Task<Result> HandleAsync(CreerFamilleCommand command, ICommandContext␣
↪context = null)
        {
            var result = Famille.CreerFamille(command.Nom);
            if(result && result is Result<NomFamille> resultFamille)
            {
                await CoreDispatcher.PublishEventAsync(new FamilleCreee(resultFamille.
↪Value));
            }
```

(suite sur la page suivante)

(suite de la page précédente)

```
            return result;
        }
    }
}
```

L'handler de la command est une classe qui implémente l'interface `ICommandHandler`, pour la commande qu'on veut gérer. Une seule méthode est à définir, `HandleAsync`. Ici, le comportement est anecdotique et se veut être pour l'exemple, sans réel impact sur le système. On ajoutera dans un prochain temps la récupération des familles depuis une source de données pour permettre au domaine de prendre la meilleure décision.

Une autre particularité est qu'un handler de command renvoie un objet de type `Result`. Cet objet n'est **PAS** là pour remplacer la notion évenementielle, mais pour avertir l'appelant de l'échec ou du succès de son appel. Dans notre cas, l'échec contient une notion métier qui peut être utile au code qui a envoyé la commande, mais cette notion de résultat est également utilisée par le système pour déterminer de la suite des actions à entreprendre.

Note : Il est possible de procéder différement ici et de ne pas retourner le `Result` obtenu par l'appel métier mais de retourner un `Result` épuré de ces notions. De la même façon, il est possible gérer des événements positifs comme des événements négatifs, mais nous aurions alors une problématique de nombre (pour chaque action/command, au minimum deux événéments : un positif et un négatif) et de pertinence (dans un système event-sourcé, les évenements négatifs n'ont aucun sens).

Il y a bien sûr, plusieurs commands pour le domaine. A titre d'exercice, et avant de consulter la solution, vous pouvez vous entrainer et créer les commandes (ainsi que les handlers) pour les actions : *AjouterPersonne*, *SupprimerFamille*.

1.16 Events

Nous avons vu comment solliciter le système pour que ce dernier entreprenne une action. Maintenant, il est nécessaire que le système réponde. En effet, nous avons envoyé la commande dans le système, et à part en debug en mettant un point d'arrêt, on a aucun retour du système, pour savoir si ça c'est bien ou mal passé, quand ça s'est terminé, etc... C'est ici que les événements domaine entre en jeu.

Un événement est généré par un agrégat, qui le stocke dans sa liste interne suite à une ou plusieurs action. Une fois ses traitements terminés, on peut lui demander de propager ses événements dans le système. En premier lieu, il faut donc créer l'événement :

```
using CQELight.Abstractions.Events;
using Geneao.Identity;

namespace Geneao.Events
{
    public sealed class FamilleCreeeEvent : BaseDomainEvent
    {
        public NomFamille NomFamille { get; private set; }

        private FamilleCreeeEvent() { }

        internal FamilleCreeeEvent(NomFamille nomFamille)
        {
            NomFamille = nomFamille;
        }

    }
}
```

La classe d'événement contient dans notre cas plus ou moins les mêmes infos que la classe de commande. Ceci s'explique par le fait que notre système, dans le cas présent, ne génère ni ne transforme aucune information (si ce n'est la conversion d'un string en `NomFamille`). Par contre, il est nécessaire de restituer les mêmes infos que la commande, ou tout du moins les infos nécessaire pour remettre le système dans un état équivalent, pour l'Event Sourcing, comme nous allons le voir par la suite.

De la même façon que pour la commande, la classe d'événement doit être sealed. Elle n'expose d'ailleurs aucun constructeur publique, car un évènement n'est envoyé que d'un et un seul contexte, mais peut être reçu par plusieurs. Evidemment, il est toujours possible de faire de la reflection pour contourner le système, mais l'idée est d'éviter les erreurs de développeurs honnêtes. Le(s) seul(s) constructeur(s) visible(nt) doit(vent) être de portée internal, car on doit permettre uniquement les objets de l'assembly de créer et d'envoyer des évènements.

Cet événement, ainsi que les événements négatifs, doivent être générés lors de la méthode `CreerFamille` de la classe `Famille`. Nous avons plusieurs choix d'implémentations. Un de ceux-ci est de conserver la méthode statique et de demander à notre agrégat de générer les événements de la demande de création. Une fois qu'on récupère l'agrégat dans notre handler, on peut utiliser le dispatcher pour envoyer les événements dans le système. Le problème avec cette méthode et qu'on ne peut se servir des événements que lorsque l'agrégat a été correctement créé. Ainsi, on se prive de la possibilité de valider niveau agrégat de la validation du nom de famille.

Une autre solution que la méthode `CreerFamille` renvoie une collection d'événements suite au traitement de la méthode. Dans notre exemple, c'est ce que nous faisons pour bien exposer la réflexion événementielle qu'il doit y avoir à l'origine.

Code à changer côté agrégat :

```
// Dans les members
internal static List<NomFamille> _nomFamilles = new List<NomFamille>();

public static Result CreerFamille(string nom, IEnumerable<Personne> personnes = null)
{
   NomFamille nomFamille = new NomFamille();
   try
   {
       nomFamille = new NomFamille(nom);
   }
   catch
   {
       return Result.Fail(FamilleNonCreeeCar.NomIncorrect);
   }
   if (_nomFamilles.Any(f => f.Value.Equals(nom, StringComparison.OrdinalIgnoreCase)))
   {
       return Result.Fail(FamilleNonCreeeCar.FamilleDejaExistante);
   }
   _nomFamilles.Add(nomFamille);
   return Result.Ok(nomFamille);
}
```

Code à changer côté handler :

```
public async Task<Result> HandleAsync(AjouterPersonneCommand command, ICommandContext␣
↪context = null)
{
   var famille = new Famille(command.NomFamille);
   var resultAjout = famille.AjouterPersonne(command.Prenom, new␣
↪InfosNaissance(command.LieuNaissance, command.DateNaissance));
   if (resultAjout)
   {
       await famille.PublishDomainEventsAsync();
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }
    return resultAjout;
}
```

En event-sourcing, les événements sont la source de données et la source de vérité. Ils sont également à la base du flux de l'application. Il est donc nécessaire de capter des événements afin de pouvoir traiter le résultat de la réaction du système, comme mettre éventuellement à jour la base de données, écrire dans un fichier, etc...

Pour ce faire, le comportement est fortement similaire à celui des commands, il faut créer un handler et agir en conséquence. Ici, nous allons en créeer l'handler de l'événement `FamilleCreeeEvent` :

```
class FamilleCreeeEventHandler : IDomainEventHandler<FamilleCreeeEvent>
{
    public Task<Result> HandleAsync(FamilleCreeeEvent domainEvent, IEventContext␣
↪context = null)
    {
        var color = Console.ForegroundColor;

        Console.ForegroundColor = ConsoleColor.DarkGreen;

        Console.WriteLine("La famille " + domainEvent.NomFamille + " a correctement␣
↪été créée dans le système.");

        Console.ForegroundColor = color;

        return Result.Ok();
    }
}
```

Note : Ici, nous n'avons pas de logique métier complexe, le système est sur-dimensionné par rapport aux besoins réel. Dans des cas métier réels complexes, cette séparation et cette granularité est généralement plus un gain qu'un frein.

Note : Les informations en cas d'échec (métier ou technique) sont transmises directement à l'appelant lorsqu'il envoie la commande dans le système. Il n'est donc pas nécessaire de créer un process à base d'événement(s) négatif(s).

1.17 Accès aux données

Pour notre système, afin d'éviter de tout recommencer à 0 à chaque démarrage, nous allons créer un système de persistance de nos données dans un fichier au format Json. Ce fichier servira également de base pour récupérer les familles existantes lors des demandes métier. Afin de pouvoir répondre à cette problématique, nous allons devoir :

1. Créer les modèles qui seront (dé)sérialisés dans notre fichier Json
2. Créer un repository accessible en lecture et en écriture, mappé sur le fichier Json
3. Ajouter les handlers correspondants aux évènements nécessitant une modification de données

1.17.1 Modèles

Un fichier Json pourrait se gérer comme une base NoSQL, nous allons stocker les familles ainsi que les personnes associées sous forme de grappe. Pour ce faire, notre modèle sera une mise à plat du contenu d'une famille

```
using CQELight.DAL.Attributes;
using CQELight.DAL.Common;
using System.Collections.Generic;

namespace Geneao.Data.Models
{
    [Table("Familles")]
    public class Famille : IPersistableEntity
    {
        [PrimaryKey]
        public string Nom { get; set; }
        public ICollection Personnes { get; set; }

       public object GetKeyValue()
           => Nom;

       public bool IsKeySet()
           => !string.IsNullOrWhiteSpace(Nom);
    }

    [Table("Personnes")]
    public class Personne : IPersistableEntity
    {
        [Column]
        public string Prenom { get; set; }
        [Column]
        public string LieuNaissance { get; set; }
        [Column]
        public DateTime DateNaissance { get; set; }
        [ForeignKey]
        public Famille Famille { get; set; }
        [Column("NomFamille"), KeyStorageOf(nameof(Famille))]
        public string Famille_Id { get; set; }

       public object GetKeyValue()
           => PersonneId;

       public bool IsKeySet()
           => PersonneId != Guid.Empty;
    }
}
```

Note : Même si, dans notre cas, les attributs sont inutiles car l'écriture est à plat dans un fichier, cela permet de migrer ultérieurement vers un autre système de persistance de données.

Afin de connaitre la totalité des possiblités offertes par CQELight pour stocker les informations sur une source de persistance, rendez-vous sur *Mapping modèle de données*.

1.17.2 Repository Json

Pour pouvoir utiliser un fichier comme source de données, nous devons définir une implémentation de repository a utiliser dans nos handlers

```
class FamilleRepository
{
    private readonly List<Famille> _familles;
    private string _filePath;

    public FileFamilleRepository()
        : this(new FileInfo("./familles.json"))
    {
    }

    public FamilleRepository(FileInfo jsonFile)
    {
        _filePath = jsonFile.FullName;
        var familles = JsonConvert.DeserializeObject<IEnumerable<Famille>>(File.
↪ReadAllText(_filePath));
        if (familles?.Any() == true)
        {
            _familles = new List<Famille>(familles);
        }
    }

    public Task<IEnumerable<Famille>> GetAllFamillesAsync()
        => Task.FromResult(_familles.AsEnumerable());

    public Task<Famille> GetFamilleByNomAsync(NomFamille nomFamille)
        => Task.FromResult(_familles.FirstOrDefault(f => f.Nom.Equals(nomFamille.
↪Value, StringComparison.OrdinalIgnoreCase)));

    public Task SauverFamilleAsync(Famille famille)
    {
        _familles.Add(famille);
        File.WriteAllText(_filePath, JsonConvert.SerializeObject(_familles));
        return Task.CompletedTask;
    }
}
```

Note : Ici nous avons passé un chemin en dur à notre constructeur par défaut sur un fichier. Il faut que ce fichier existe. Nous devons donc rajouter au début de notre application un test pour voir si le fichier existe, et si non, le créer au format json avec un contenu vide (donc un fichier avec le contenu "[]", sans les apostrophes)

1.17.3 Changement des handlers

Maintenant, il est nécessaire de modifier nos handlers (et notre agrégat famille) pour récupérer les informations depuis le fichier, tout comme il est nécessaire de créer des handlers d'évènements pour mettre à jour le fichier lorsque les opérations ont été réalisées avec succès.

Nous allons commencer par modifier le handler d'évenement de création de famille

```
class FamilleCreeeEventHandler : IDomainEventHandler<FamilleCreeeEvent>
{
    public async Task<Result> HandleAsync(FamilleCreeeEvent domainEvent,␣
↪IEventContext context = null)
    {
        var color = Console.ForegroundColor;
```

(suite sur la page suivante)

(suite de la page précédente)

```
        try
        {
            await new FileFamilleRepository().SauverFamilleAsync(new Data.Models.
↪Famille
            {
                Nom = domainEvent.NomFamille.Value
            }).ConfigureAwait(false);

            Console.ForegroundColor = ConsoleColor.DarkGreen;
            Console.WriteLine("La famille " + domainEvent.NomFamille.Value + " a␣
↪correctement" +
                " été créée dans le système.");
        }
        catch (Exception e)
        {
            Console.ForegroundColor = ConsoleColor.DarkRed;
            Console.WriteLine("La famille " + domainEvent.NomFamille.Value + " n'a␣
↪pas pu être" +
                " créée dans le système.");
            Console.WriteLine(e.ToString());
            return Result.Fail();
        }
        finally
        {
            Console.ForegroundColor = color;
        }
        return Result.Ok();
    }
}
```

Notre handler de `FamilleCreee` fait maintenant plus que simplement afficher sur la console comme quoi l'opération a réussi ou non au niveau domaine (et donc entièrement en mémoire). Une fois l'opération réussie, la famille est persistée pour les prochaines exécution. Notre domaine reste responsable de la cohérence du système. Cependant, il faut que le domaine soit au courant des informations qui ont été persistées. C'est le rôle du CommandHandler de palier à cette problématique d'infrastructure, il se doit donc de récupérer les informations depuis la persistance et restituer les informations dans le domaine

```
class CreerFamilleCommandHandler : ICommandHandler<CreerFamilleCommand>
{
    public async Task<Result> HandleAsync(CreerFamilleCommand command,␣
↪ICommandContext context = null)
    {

          Famille._nomFamilles = (await new FileFamilleRepository().
↪GetAllFamillesAsync()
          .ConfigureAwait(false)).Select(f => new Identity.NomFamille(f.Nom)).
↪ToList();

        var result = Famille.CreerFamille(command.Nom);
        if(result && result is Result<NomFamille> resultFamille)
        {
            await CoreDispatcher.PublishEventAsync(new FamilleCreee(resultFamille.
↪Value));
            return Result.Fail();
        }
        return result;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }
}
```

Notre agrégat est donc restauré à un état où il connait le contenu des données de la persistance afin de prendre la bonne décision pour l'ensemble du système (parce qu'ici, notre source de vérité est le fichier qui contient l'ensemble des familles). Lors de nos différentes exécutions, on retrouvera l'ensemble de nos familles de cette façon. Grâce au repository, on peut également se permettre d'implémenter une fonction d'affichage de la liste des familles présentes dans le système.

Cependant, il y a un problème majeur avec le code ainsi produit, c'est qu'il ne peut fonctionner qu'avec le repository de fichier. Le jour où, pour des raisons de performances ou de nécessité de stockage, il est nécessaire de stocker les informations en base de données, il sera nécessaire de rechercher tous les appels au FileFamilleRepository pour les remplacer. Et si un retour arrière ou un autre changement est nécessaire, le problème se répètera encore et encore. La solution pour ça consistera à travailler avec des abstractions au niveau code et de laisser CQELight se charger de résoudre les implémentations, comme nous allons le voir dans la partie sur l'IoC.

1.18 Inversion of Control

Nous avons vu dans la précédente étape la création d'un repository. Cette classe nous permet de sauver nos familles dans un fichier texte et de les récupérer au lancement. Il y a cependant un problème avec le code présent : l'appel au repository de familles sous forme de fichier est fait en dur. Cela signifie en substance qu'il est difficile de changer de technologie de persitance au fil de l'application sans réécrire le code, tout en sachant que normalement cette portion de code en production est testée et approuvée. Il faut revoir notre code pour travailler avec des abstractions. Et tant qu'à le revoir, autant respecter les bonnes pratiques avec l'IoC : procéder à l'injection par constructeur (ce qui permet d'afficher sur notre API publique qu'une instance répondant à cette interface est nécessaire pour que la classe fonctionne correctement).

Il est également nécessaire de définir l'interface des méthodes communes qui doivent être implémentées par chaque repository (et bien entendu rajouter dans notre FileFamilleRepository le fait qu'il implémente cette interface)

```
public interface IFamilleRepository
{
        Task SauverFamilleAsync(Famille famille);
        Task<Famille> GetFamilleByNomAsync(NomFamille nomFamille);
        Task<IEnumerable<Famille>> GetAllFamillesAsync();
}
```

Nous allons donc devoir modifier notre handler d'évènement (pour gérer correctement l'évènement FamilleCreee) pour supprimer l'appel qui se fait directement sur `FileFamilleRepository` :

```
class FamilleCreeeEventHandler : IDomainEventHandler<FamilleCreee>, IAutoRegisterType
{
    private readonly IFamilleRepository _familleRepository;

    public FamilleCreeeEventHandler(IFamilleRepository familleRepository)
    {
        _familleRepository = familleRepository ?? throw new␣
↪ArgumentNullException(nameof(familleRepository));
    }

    public async Task<Result> HandleAsync(FamilleCreee domainEvent, IEventContext␣
↪context = null)
    {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        var color = Console.ForegroundColor;
        try
        {
            await _familleRepository.SauverFamilleAsync(new Data.Models.Famille
            {
                Nom = domainEvent.NomFamille.Value
            }).ConfigureAwait(false);

            Console.ForegroundColor = ConsoleColor.DarkGreen;
            Console.WriteLine("La famille " + domainEvent.NomFamille.Value + " a␣
↪correctement" +
            " été créée dans le système.");

        }
        catch (Exception e)
        {
            Console.ForegroundColor = ConsoleColor.DarkRed;
            Console.WriteLine("La famille " + domainEvent.NomFamille.Value + " n'a␣
↪pas pu être" +
            " créée dans le système.");
            Console.WriteLine(e.ToString());
            return Result.Fail();
        }
        finally
        {
            Console.ForegroundColor = color;
        }
        return Result.Ok();
    }
}
```

Notre code est retouché pour permettre de travailler avec des abstractions. Mais en l'absence de configuration au niveau du système de CQELight d'un fonctionnement IoC, les handlers ne seront plus appelés, rendant notre système inopérant. Pour ce faire, nous devons, à l'instar du bus in-memory, installer un plugin nous permettant de gérer l'IoC et le configurer.

Note : CQELight a fait le choix de n'embarquer aucun module d'IoC ni de développer son propre système afin de laisser le choix aux développeurs de l'outil à utiliser. Le système fonctionne sans IoC tant que la logique des constructeurs sans paramètres est respectée. Si on choisit de l'appliquer à nos handlers si dessus, il faudrait un constructeur sans paramètre qui appelle le constructeur avec paramètre avec l'instance par défaut.

L'un des container les plus utilisés sur le marché est Autofac. CQELight mets à disposition un plugin pour ce dernier. Il suffit d'installer le package correspondant pour commencer à l'utiliser : `CQELight.IoC.Autofac`. Les spécificités de ce plugin sont décrites dans la page dédié et ne seront pas explorées ici.

Il est nécessaire de retoucher notre `FileFamilleRepository` afin d'utiliser la possibilité qu'offre CQELight d'automatiquement enregistrer le type dans le container

```
class FileFamilleRepository : IFamilleRepository, IAutoRegisterTypeSingleInstance
{
    private readonly ConcurrentBag<Famille> _familles = new ConcurrentBag<Famille>();
    private string _filePath;

    public FileFamilleRepository()
        : this(new FileInfo("./familles.json"))
```

(suite sur la page suivante)

(suite de la page précédente)

```
    {

    }

    public FileFamilleRepository(FileInfo jsonFile)
    {
        _filePath = jsonFile.FullName;
        var familles = JsonConvert.DeserializeObject<IEnumerable<Famille>>(File.
↪ReadAllText(_filePath));
        if (familles?.Any() == true)
        {
            _familles = new ConcurrentBag<Famille>(familles);
        }
    }

    public Task<IEnumerable<Famille>> GetAllFamillesAsync()
        => Task.FromResult(_familles.AsEnumerable());

    public Task<Famille> GetFamilleByNomAsync(NomFamille nomFamille)
        => Task.FromResult(_familles.FirstOrDefault(f => f.Nom.Equals(nomFamille.
↪Value, StringComparison.OrdinalIgnoreCase)));

    public Task SauverFamilleAsync(Famille famille)
    {
        _familles.Add(famille);
        File.WriteAllText(_filePath, JsonConvert.SerializeObject(_familles));
        return Task.CompletedTask;
    }
}
```

Note : Attention, avec cette méthode, en cas de création d'un nouveau repository, il sera dès lors nécessaire de supprimer l'interface IAutoRegisterTypeSingleInstance du FileSystemRepository pour la mettre sur notre nouvelle implémentation pour que ça soit celle par défaut. D'autre part, la notion de singleton n'a de sens que pour notre repository de fichier car celui-ci utilise une liste mémoire pour gérer le contenu. Le fait d'avoir un singleton oblige également à rendre notre code sécuritaire sur les accès concurrentiels (utilisation d'un ConcurrentBag).

Note : Le fait d'utiliser `IAutoRegisterType` enregistre le type dans le container par défaut. Ainsi, le container tentera de résoudre chacun des paramètres d'un constructeur, ou utilisera le constructeur sans paramètre s'il n'y arrive pas. Dans notre cas, on a un constructeur qui utilise un fichier par défaut. Cependant, si l'on aurait voulu fournir un autre fichier ou avoir une logique métier du fichier à utiliser par le repository, il aurait été nécessaire de faire un enregistrement manuel dans le container.

On va rajouter au bootstrapper de notre application le fait que le système doit utiliser Autofac comme container IoC

```
new Bootstrapper()
.UseInMemoryEventBus()
.UseInMemoryCommandBus()
.UseAutofacAsIoC(c => {
    //Les enregistrements manuels se font ici
})
.Bootstrapp();
```

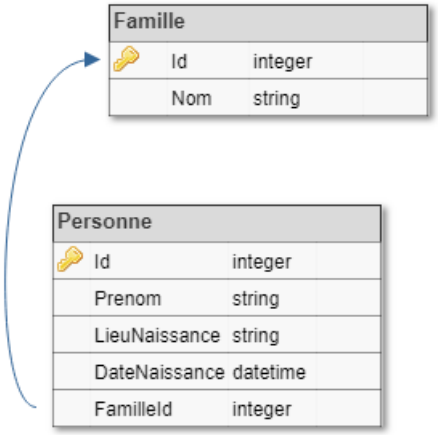
Dès que ces opérations sont réalisées, le système fonctionne de façon totalement similaire à précédemment, mais nous

offre la possibilité de fournir une autre implémentation pour le `IFamilleRepository`. A ce titre, comme exercice, vous pouvez essayer de créer un repository qui utilise Entity Framework Core pour stocker les informations dans une base de données et de donner le choix à l'utilisateur au lancement de l'application de quel type de persistance il veut bénéficier.

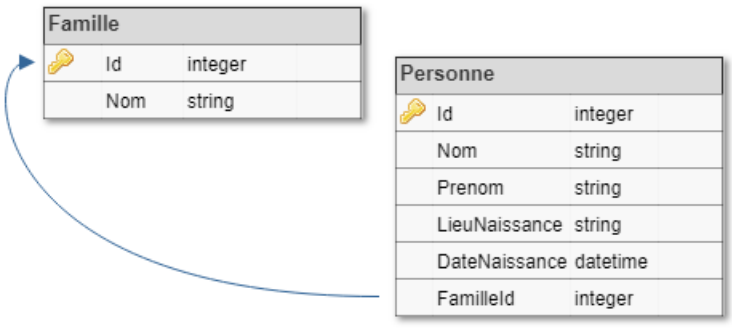
1.19 Queries

Dans l'état actuel de notre système, il n'est possible que d'ajouter des familles et non pas de les lister, alors que le repository le permet (méthode `GetAllFamillesAsync`). Pourtant, les données existent et ne demandent qu'à être consultées. Il s'agit de l'implémentation de la couche Query (pattern CQRS). L'idée principale derrière le CQRS est l'extrême performance en lecture, car un logiciel métier passe le plus clair de son temps à lire des données plutôt qu'à écrire.

Si notre logiciel est amené à évoluer vers une base de données, il faudra que cette base respecte le principe CQRS et soit une base optimisée pour la lecture. Dans le cas de notre exemple, en se focalisant sur les problématiques BDD, on imagine facilement une structure base de données comme cela :



Ce schéma de base de données normalisé est très optimisé pour la gain d'espace disque. Cependant, les principes de normalisation remonte à une époque où l'espace disque était plus cher et très rare. De nos jours, la tendance n'est plus la même, on a énormément d'espace disque, le point bloquant étant les ressources de calcul (CPU et RAM). On va donc « dénormaliser » la base de données, pour mettre le maximum de données à plat, afin d'éviter de trop solliciter les moteurs relationnels et gagner en temps de traitement. Notre schéma ressemblera donc à ceci :



La différence réside dans le fait que le nom de famille est dupliqué dans chaque ligne de la table personne, évitant ainsi d'avoir à faire une jointure sur la table Famille. Se pose dès lors la question de la nécessité d'une telle table. Avec une réflexion purement SQL, on se contenterait d'indexer la colonne "Nom" et de faire un select distinct. Le problème, c'est que si cette opération est récurrente, on sera moins performant qu'un `SELECT *`.

Cela est effectivement insignifiant au niveau de notre exemple, mais sur des données beaucoup plus massive, les gains de temps peuvent être énormes.

Bien évidemment, la mise à plat n'est pas toujours possible ou facile à concevoir, et il existera des situations où il faudra faire des jointures. L'idéal est de réduire au maximum les traitements à faire côté base de données, pour distribuer l'information le plus rapidement possible au contexte demandant.

Pour pouvoir gérer ces problématique, il va falloir que l'on mette en place une couche Query. Le but de cette couche Query est de créer des sources de données optimisées pour la lecture, de fournir une API pour les lires, et se charger de leur mise à jour. La couche Query doit renvoyer des ViewModels, des DTO (Data-Transfer-Object) qui restitue les données ;

Il est important de passer par la case « DTO », afin d'avoir le formatage de données voulu, et d'éviter de renvoyer les objets de mappings/DAL. Cela peut amener à avoir plusieurs queries sur une même table, qui retourne chacune un petit bout uniquement, et c'est très bien comme cela. Encore une fois, on cherche à optimiser les lectures, on procèdera donc à la mise en cache et tout autre sorte de mécanisme pour gagner en performance.

Alors que précédemment on utilisait directement notre repository pour lire la liste des familles, on passera dorénavant par une query. Le but des query est d'encapsuler totalement le repository et les accès aux données pour les optimiser (et ne plus laisser cette responsabilité à l'appelant)

```
public class FamilleListItem
{
    public string Nom { get; internal set; }

    internal FamilleListItem() { }
}
```

```
public interface IRecupererListeFamille : IQuery<IEnumerable<FamilleListItem>> { }
class RecupererListeFamille : IRecupererListeFamille, IAutoRegisterType
{

    private static ConcurrentBag<string> s_Cache
        = new ConcurrentBag<string>();

    private readonly IFamilleRepository _familleRepository;

    public RecupererListeFamille(IFamilleRepository familleRepository)
    {
        _familleRepository = familleRepository ?? throw new␣
↪ArgumentNullException(nameof(familleRepository));
    }

    public async Task<IEnumerable<FamilleListItem>> ExecuteQueryAsync()
    {
        if (s_Cache.IsEmpty)
        {
            var allFamilles = (await _familleRepository.GetAllFamillesAsync().
↪ConfigureAwait(false)).Select(f => f.Nom);
        }
        return s_Cache.Select(v => new FamilleListItem { Nom = v });
    }
}
```

On déclare une interface publique de typage fort qui déclare les entrées/sorties de la query, pour respecter le dernier principe SOLID. Nous avons également choisi l'impémentation internal, encore une fois pour éviter les erreurs de développements qui prendrait le raccourci de travailler avec l'implémentation plutôt que l'abstraction. Cela permettra à notre système de faire évoluer les bases de lecture de façon indolore.

Dorénavant, pour liste les familles, on passera par `IRecupererListeFamille` au lieu d'utiliser le repository.

Note : Il est même recommandé de passer en internal toute le contenu de la couche persistance et d'autoriser uniquement la couche Query à les voir depuis l'extérieur (avec InternalsVisibleTo).

Il reste cependant un problème avec cet exemple : si l'on ajoute une nouvelle famille dans le système, il est nécessaire de le redémarrer pour la voir apparaître car le cache prends le pas sur la lecture. Il faut donc procéder à l'invalidation à la base d'événements. Dans notre cas, on a deux possibilité : mettre à jour le cache ou l'invalider. Le plus simple est l'ajout

```
class FamilleCreeeInvalider : IDomainEventHandler<FamilleCreee>
{
    public Task HandleAsync(FamilleCreee domainEvent, IEventContext context = null)
    {
        RecupererListeFamille.AjouterFamilleAuCache(domainEvent.NomFamille.Value);
        return Task.CompletedTask;
    }
}
public interface IRecupererListeFamille : IQuery<IEnumerable<FamilleListItem>> { }
class RecupererListeFamille : IRecupererListeFamille
{
        internal static void AjouterFamilleAuCache(string nom) => s_Cache.Add(nom);

    private static ConcurrentBag<string> s_Cache
        = new ConcurrentBag<string>();

    private readonly IFamilleRepository _familleRepository;

    public RecupererListeFamille(IFamilleRepository familleRepository)
    {
        _familleRepository = familleRepository ?? throw new␣
↪ArgumentNullException(nameof(familleRepository));
    }

    public async Task<IEnumerable<FamilleListItem>> ExecuteQueryAsync()
    {
        if (s_Cache.IsEmpty)
        {
            var allFamilles = (await _familleRepository.GetAllFamillesAsync().
↪ConfigureAwait(false)).Select(f => f.Nom);
        }
        return s_Cache.Select(v => new FamilleListItem { Nom = v });
    }
}
```

A titre d'exercice, vous pouvez créer les queries de récupération des personnes, et même éventuellement rajouter des paramètres (par exemple une date de naissance minimale, ou juste ceux qui sont vivants). Le nombre de queries importe peu et doit répondre aux cas métiers. De la même façon, au lieu d'essayer de toujours essayer de fonctionner avec le cas existant, considérez également la création de table/vue pour optimiser vos traitements.

1.20 Event sourcing

Au jour d'aujourd'hui, notre système n'utilise qu'une seule et même source de données pour la lecture que pour la prise de décision. On a vu précédemment qu'on a commencé à faire des choix pour optimiser la lecture. Sauf qu'en

contre-partie, on optimise pas l'aspect écriture. De la même façon, les données dans notre fichier font état des choses à instant T, on ne sait pas ce qu'il s'est passé à T-1 ou depuis la genèse du système.

Pourtant notre système est déjà prêt pour ce type de fonctionnement, car chacune des actions que l'on a fait génere un ou plusieurs événements qui font état de l'histoire du système qui s'est déroulée. Il suffit de sauvegarder ces événements dans un event-store afin de pouvoir les récupérer et les rejouer si nécessaire. CQELight fournit plusieurs implémentations d'event-store pour accomplir cet objectif.

Dans le cas présent, on va prendre le plus simple à mettre en place : un event-store avec Entity Framework Core et SQLite, mais le fonctionnement global reste similaire peu importe le provider choisi

```
ew Bootstrapper()
   .UseInMemoryEventBus()
   .UseInMemoryCommandBus()
   .UseAutofacAsIoC(c =>
   {
   })
   .UseEFCoreAsEventStore(
   new CQELight.EventStore.EFCore.EFEventStoreOptions(
       c => c.UseSqlite("FileName=events.db", opts => opts.
↪MigrationsAssembly(typeof(Program).Assembly.GetName).Name))))
   .Bootstrapp();
```

Il est dès lors nécessaire de générer la migration EntityFramework Core. Pour ce faire, il faut créer une classe qui permet de définir la façon d'obtenir un contexte au moment du design

```
public class EventStoreDbContextCreator : IDesignTimeDbContextFactory
↪<EventStoreDbContext>
{
    public EventStoreDbContext CreateDbContext(string[] args)
    {
        return new EventStoreDbContext(new DbContextOptionsBuilder
↪<EventStoreDbContext>()
                    .UseSqlite("FileName=events.db", opts => opts.
↪MigrationsAssembly(typeofEventStoreDbContextCreator).Assembly.GetName().Name))
                    .Options, SnapshotEventsArchiveBehavior.Delete);
    }
}
```

Cette classe sera utilisé par le CLI d'EF Core afin d'avoir accès au contexte pour générer la migration. Dès lors, il suffit d'appeler le CLI pour générer une migration : dotnet ef migrations add EventStoreMigration -c EventStoreDbContext. La migration sera générée dans le projet.

Note : Il n'est pas nécessaire de faire appel au CLI pour demander un database update ou de faire une migration par code, le bootstrapper se charger de récupérer la migration et de l'appliquer si vous précisez bien l'option MigrationsAssembly dans la connectionString.

Note : Ces blocs de code ne sont donnés que pour l'exemple. Dans un environnement de production, il est préférable de stocker la chaine de connexion et le paramétrage à un endroit plus sécurisé, comme une variable d'environnement ou un fichier de configuration.

En ayant suivant les étapes précédentes, on arrive donc à avoir un système fonctionnel avec un event-store, qui capte et enregistre les événements précédemment créés. Le souci, c'est que l'état actuel des événements ne permettent pas de remettre le système dans une condition fonctionnelle, car les événements ne sont pas liés à une identité d'aggrégat. Il faut modifier légérement l'événement de création :

```
public sealed class FamilleCreee : BaseDomainEvent
{

    public NomFamille NomFamille { get; private set; }

    private FamilleCreee() { }

    internal FamilleCreee(NomFamille nomFamille)
    {
        NomFamille = nomFamille;
        AggregateId = nomFamille;
        AggregateType = typeof(Famille);
    }

}
```

On rajoute dans l'événement `FamilleCreee` le type et l'id de l'agrégat pour que ce dernier puisse récupérer les événements le concernant.

Note : Cette modification n'est nécessaire que pour les événements qui ne sont pas enregistrés et publiés depuis une instance d'aggrégat, car si c'est le cas, le framework CQELight est capable de renseigner ces informations automatiquement, comme lors de l'action `AjouterPersonne`.

Dès que cette modification est appliquée, si on regarde la BDD, on constate qu'une ligne est ajoutée dans la table Event chaque fois qu'un événement est publié. Ces événements consitue la base en écriture dans un modèle CQRS, et doivent être utilisé par notre aggrégat chaque fois qu'une action est demandée. Il faut donc modifier notre agrgégat pour le transformer en aggrégat « event-sourcé », et ceci en deux actions :

— Implémenter l'interface `IEventSourcedAggregate` (dans le namespace `CQELight.Abstractions.EventStore.Interfaces`)
— Aggrémenter l'objet `FamilleState` de handlers capable de gérer les événements étant arrivés

1.20.1 Gestion de l'état de l'aggrégat

En modifiant notre `AggregateState`, on va lui ajouter la possibilités de savoir comment réagir aux événement pour gérer la réhydratation depuis la base, par le biais de la méthode `AddHandler`. Cette méthode défini l'extraction des informations depuis les événements vers la classe d'état elle-même. Etant donné que l'on agit en réhydratation, on s'assure que l'état reste cohérent, en définissant les setters privés :

```
class Famille : AggregateRoot<NomFamille>
{
    private FamilleState _state;

    private class FamilleState : AggregateState
    {

        public List<Personne> Personnes { get; private set; }
        public string Nom { get; private set; }

        public FamilleState()
        {
            Personnes = new List<Personne>();
            AddHandler<FamilleCreee>(FamilleCree);
        }
```

(suite sur la page suivante)

(suite de la page précédente)

```
        private void FamilleCree(FamilleCreee obj)
        {
            Nom = obj.NomFamille.Value;
            _nomFamilles.Add(obj.NomFamille);

        }
    }
    [...]
}
```

1.20.2 Gestion d'un aggrégat event-sourcé

Il est nécessaire d'implémenter l'interface `IEventSourcedAggregate` afin de donner au système la visibilité sur les possibilités de cet aggrégat. L'implémentation de cette interface nécessite de redéfinir le comportement RehydrateState, qui permet à l'event-store de réhydrater l'aggrégat simplement.

Note : Il est possible d'hériter d'une classe qui permet cela de façon automatique : `EventSourcedAggregate`. Cette classe nécessite cependant de préciser le type de state que l'aggrégat va gérer, impliquant le fait que cette classe doit avoir une visibilité publique. La difficulté va résider dans la vigilance nécessaire pour consever le périmètre de la responsabilité de cette classe, à savoir conserver la cohésion des données.

```
class Famille : AggregateRoot<NomFamille>, IEventSourcedAggregate
{

        public void RehydrateState(IEnumerable<IDomainEvent> events)
        {
            _state.ApplyRange(events);
            Id = _state.Nom;
        }

    [...]
}
```

1.20.3 Assemblage des élements

La problématique de réhydratation est une problématique infrastructurale, et revient donc aux command handlers. Afin d'illustrer un cas, on va l'appliquer sur la gestion de la command AjouterPersonne. Cela va se concrétiser en demandant une injection d'un IAggregateEventStore, afin de pouvoir récupérer une aggrégat totalement réhydraté

```
class AjouterPersonneCommandHandler : ICommandHandler<AjouterPersonneCommand>,␣
↪IAutoRegisterType
{
    private readonly IAggregateEventStore _eventStore;

    public AjouterPersonneCommandHandler(IAggregateEventStore eventStore)
    {
        _eventStore = eventStore;
    }
    public async Task<Result> HandleAsync(AjouterPersonneCommand command,␣
↪ICommandContext context = null)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    {
        var famille = await _eventStore.GetRehydratedAggregateAsync<Famille>(command.
↪NomFamille);
        famille.AjouterPersonne(command.Prenom, new InfosNaissance(command.
↪LieuNaissance, command.DateNaissance));
        await famille.PublishDomainEventsAsync();
        return Result.Ok();
    }
}
```

La boucle est bouclée, on gère le circuit des événements depuis l'envoi jusqu'à l'utilisation pour la réhydratation. En utilisant la réhydration pour effectuer une commande, on s'assure d'utiliser le fil de l'histoire pour prendre la meilleure décision possible, et pas uniquement un état à un instant T. Il nous reste maintenant à sécuriser notre code à l'aide de tests automatisés.

1.21 Tests unitaires et fonctionnels

Note : Chez Hybrid Technologies Solutions, nous privilégions les développements qui suivent la logique TDD (Test Driven Development), qui consiste à écrire les tests avant l'implémentation. Cette logique ayant fait ses preuves, elle n'est plus à démontrer. Cependant, dans le cadre de cet exercice, les tests arrivent en fin de course, car l'objectif n'est pas d'apprendre le TDD (bien que si vous le désirez, Hybrid Technologies Solutions peut vous former à cela) mais bien de découvrir les possibilités de CQELight. Les tests arrivent donc à cette étape uniquement pour cette raison, et aucune autre.

Comme nous l'avons vu sur la page concernant les tests unitaires et d'intégration, CQELight fournit un package permettant d'écrire des tests automatisés et de vérifier la logique métier. Dans le cadre de notre exemple, nous allons écrire deux tests, un unitaire et un d'intégration, pour démontrer le principe.

La totalité du code ne sera pas couverte, mais il ne tient qu'à vous de poursuivre l'exercice et de tenter d'en couvrir un maximum. A noter cependant qu'il est pertinent de focaliser les efforts d'écriture et de maintenance des tests sur la partie métier de votre application et non la partie technique, celle-ci reposant généralement sur des outils étant déjà testés et benchmarkés.

1.21.1 Test unitaire

Afin d'écrire un test pertinent sur notre exemple, nous allons vérifier le comportement de la méthode AjouterPersonne de l'aggrégat Famille, s'assurer de la récupération d'un événement ou d'un résultat négatif au sens métier. Il y a deux possibilités pour tester ce comportement : soit directement auprès de l'aggrégat, soit auprès d'un command handler.

Dans le contexte d'un test unitaire, qui doit être détaché de toute problématique infrastructurale et technique, il est pertinent de faire cet appel sur l'aggrégat lui-même et de vérifier le résultat, et ce pour deux raisons :

1. Cela permet d'écrire un test qui s'exécutera très rapidement et ne concernera que la logique métier
2. On embarque pas dans notre test des notions de mock pour simuler certains comportement techniques particuliers, on reste au niveau d'une fonction mathématique pure

En faisant cela, non seulement on sécurire notre logique métier, mais on fourni également une documentation implicite pour les autres développeurs (ils voient un exemple concret d'appel). En évitant de descendre trop bas dans l'implémentation, on s'assure également de garder une API utilisable. Dans le cas présent, l'existance d'un TU sur l'API AjouterPersonne permet le refactoring au sein de l'implémentation, mais « bloque » la signature de la méthode, qui peut être utilisée ailleurs.

Note : Cela amène également le sujet de l'importance de conserver des API rétro-compatibles si vous n'avez pas la main sur l'ensemble des services utilisant vos APIs. Avoir un ou plusieurs tests qui couvrent ces API permettra d'éviter de casser une API utilisée par d'autres services. La notion de Command est là pour abstraire ce problème et permettre à votre domaine d'évoluer sans impact, d'où il est important d'utiliser ce système plutôt que de faire appel directement à l'aggrégat, dans la mesure du possible.

La première étape consiste à créer un projet de tests automatisés. Nous utilisons xUnit pour cela, mais vous êtes libre d'utiliser le framework de votre choix. A ce projet, nous allons ajouter le package CQELight.TestFramework. Notre classe doit hériter de BaseUnitTestClass pour profiter de la panoplie d'outils à disposition. Ecrivons notre premier test qui couvre le cas optimal fonctionnel (ajout avec succès d'une personne qui n'existe pas dans la famille).

```
public class FamilleTests : BaseUnitTestClass
{
    [Fact]
    public void Ajouter_Personne_Should_Create_Event_PersonneAjoute()
    {
        var familleResult = Famille.CreerFamille("UnitTest");
        familleResult.Should().BeOfType<Result<NomFamille>>();

        var famille = new Famille("UnitTest");

        var result = famille.AjouterPersonne("First",
            new InfosNaissance("Paris", new DateTime(1965, 12, 03)));

        result.IsSuccess.Should().BeTrue();
        famille.DomainEvents.Should().HaveCount(1);
        famille.DomainEvents.First().Should().BeOfType<PersonneAjoutee>();
        var evt = famille.DomainEvents.First().As<PersonneAjoutee>();
        evt.Prenom.Should().Be("First");
        evt.DateNaissance.Should().BeSameDateAs(new DateTime(1965, 12, 03));
        evt.LieuNaissance.Should().Be("Paris");

    }
}
```

Ce test suit une logique simple : création d'une famille dans le système, récupération de l'aggrégat de famille, ajout d'une personne, vérification que tout est ok. Notre premier test est écrit. Il manque maintenant les tests « négatifs ». Un exemple de ceux-ci peut être

```
[Fact]
public void Ajouter_Personne_Already_Exists_Should_Returns_Result_Fail()
{
    var familleResult = Famille.CreerFamille("UnitTest");
    familleResult.Should().BeOfType<Result<NomFamille>>();

    var famille = new Famille("UnitTest");

    famille.AjouterPersonne("First",
        new InfosNaissance("Paris", new DateTime(1965, 12, 03)));

    var result = famille.AjouterPersonne("First",
        new InfosNaissance("Paris", new DateTime(1965, 12, 03)));

    result.IsSuccess.Should().BeFalse();
    result.Should().BeOfType<Result<PersonneNonAjouteeCar>>();
```

(suite sur la page suivante)

(suite de la page précédente)

```
    var raison = result.As<Result<PersonneNonAjouteeCar>>().Value;
    raison.Should().Be(PersonneNonAjouteeCar.PersonneExistante);
}
```

Lors de l'exécution des tests, un des deux tests ne passent pas, car en effet, la famille existe déjà dans la variable statique `_nomFamilles` (stockée dans l'aggrégat `Famille`)! Pour palier à ce problème, nous avons plusieurs solutions. Une d'entre elle consisterait à exposer les variables de portée internal à notre assembly de tests. Une autre consisterait à utiliser un autre nom de famille. Pour résoudre vite ce problème, nous allons déplacer la création de famille dans une méthode d'initialisation de notre constructeur

```
private static bool s_Init = false;
public FamilleTests()
{
    if(!s_Init)
    {
        Famille.CreerFamille("UnitTest");
        s_Init = true;
    }
}
```

Nos deux tests passent avec succès. Maintenant que vous avez la logique, il devient très facile d'écrire les tests pour le cas PrenomInvalide. Attention à un point dans ce cas précis : si les informations de naissance sont mal renseignées, le test échoue mais pas à cause d'une logique implémentée dans l'aggrégat mais dans l'entité personne. Le choix vous appartient d'écrire le test au niveau entité ou aggrégat, il faut juste garder à l'esprit de ne pas bloquer le refactoring en descendant trop bas.

1.21.2 Test d'intégration

Dans une logique de test d'intégration, il convient de mettre en place la structure pour s'assurer que la totalité des élements s'assemblent bien. Nous allons tester la même chose que précédemment, mais en mode intégration. Il est nécessaire de créer un nouveau projet de tests automatisés afin d'y implémenter notre test d'intégration, en suivant la convention de nommage : xxxx.Integration.Tests. Cette règle a été définie afin de clairement séparer les tests unitaires (répétables et intégrables dans un pipeline devops) des tests d'intégrations (lancement moins fréquent et dépendant de contraintes d'environnement rendant l'automatisation moins évidente).

Cette règle peut-être contournée, mais il est préférable de suivre la recommandation pour profiter au maximum des optimisations prévues pour chaque type de test

```
[Fact]
public async Task Ajouter_Personne_Should_Publish_Event_PersonneAjoute()
{
    new Bootstrapper()
        .UseInMemoryEventBus()
        .UseInMemoryCommandBus()
        .UseAutofacAsIoC(_ => { })
        .UseEFCoreAsEventStore(
            new CQELight.EventStore.EFCore.EFEventStoreOptions(
                c => c.UseSqlite("FileName=events_tests.db", opts => opts.
↪MigrationsAssembly(typeof(FamilleIntegrationTests).Assembly.GetName).Name)),
                archiveBehavior: CQELight.EventStore.SnapshotEventsArchiveBehavior.
↪Delete))
        .Bootstrapp();
```

(suite sur la page suivante)

(suite de la page précédente)

```
    await CoreDispatcher.DispatchCommandAsync(new CreerFamilleCommand("UnitTest"));

    var command = new AjouterPersonneCommand("UnitTest", "First", "Paris", new␣
↪DateTime(1965, 12, 03));

    var evt = await Test.WhenAsync(() => CoreDispatcher.DispatchCommandAsync(command))
        .ThenEventShouldBeRaised();

    evt.Prenom.Should().Be("First");
    evt.DateNaissance.Should().BeSameDateAs(new DateTime(1965, 12, 03));
    evt.LieuNaissance.Should().Be("Paris");

}
```

Note : Il est nécessaire de copier ou d'ajouter les migrations pour l'event store dans le projet pour que l'intégration puisse se faire de part en part. On constate la présence du bootstrapper (nécessaire pour mettre en place l'infrastructure) et l'utilisation du framework de test (avec la méthode `Test.WhenAsync`).

Comme cela se remarque facilement, le test d'intégration est plus lourd à mettre en place et plus long à l'exécution, c'est pourquoi il est recommandé de prioriser les tests unitaires lorsqu'il convient de tester le métier. Cependant, il n'en reste pas moins intéressant d'en avoir quelques-uns pour sécuriser ce qui peut être automisé.

Vous avez dorénavant la possibilité d'écrire des tests pour votre code métier !

1.22 Finalisation

En conclusion, nous avons vu comment, en partant d'un besoin initial, nous avons pu utiliser le framework CQELight afin de créer un système complet qui suit le principe du Domain Driven Design, du pattern CQRS et avec l'implémentation de l'Event Sourcing.

Chaque élément technique peut être remplacé par un autre plugin afin d'explorer les possibilités de CQELight (par exemple, utilisation de MongoDb comme provider d'Event Store). Vous avez également une base de travail si vous désirez créer vos propres extensions.

De la même façon, il ne tient qu'à vous d'agrémenter cet exemple pour le rendre plus complet au sein métier, par exemple implémenter la gestion de la parenté, changer l'interface graphique pour utiliser du web, etc.

Le code source de cet exemple est disponible sur notre repository GitHub à cette adresse. Vous avez aussi à disposition la transcription de ce cours en vidéo disponible sur Youtube à cette adresse.

1.23 IoC avec Autofac

Autofac est une libraire d'IoC très puissante et très activement maintenue et évoluée par la communauté. Le web regorge de documentation, d'exemples et d'informations à son sujet.

Pour utiliser Autofac, il faut ajouter le package `CQELight.IoC.Autofac` à votre projet. Le package est disponible sur NuGet.

Cette extension s'utilise comme toutes les autres, et s'utilise en appelant la méthode d'extension dédiée sur le bootstrapper. Il y a deux overloads de cette méthode :

— La première prends directement votre instance de `ContainerBuilder`, dans lequel vous aurez défini tous vos enregistrements et y ajoutera les types du système pouvant être utilisé par lui, et construira le Container en fin de `Bootstrapp()`
— La seconde prends en paramètre une action d'enregistrement sur un `ContainerBuilder` qui sera créé par le système, afin que vous puissiez ajouter à celui-ci vos propres enregistrements

De même, comme chaque module d'IoC, il faut pouvoir traiter les interfaces d'enregistrements automatique `IAutoRegisterType` et `IAutoRegisterTypeSingleInstance`. Le module Autofac va rechercher dans l'ensemble des types du système, c'est pourquoi un paramètre permet d'exclure des assemblies de la recherche.

```
// With lambda registration
new Bootstrapper().UseAutofacAsIoC(
    containerBuilder => { containerBuilder.RegisterType<MyClass>().
↪AsImplementedInterfaces(); }
// Excluding DLLs from searching to enhance performances (it's a contains searching)
    "CQELight", "Microsoft", "System")

// With container
var containerBuilder = new ContainerBuilder();
containerBuilder.RegisterType<MyClass>().AsImplementedInterfaces();
new Bootsrapper().UseAutofacAsIoC(containerBuilder);
```

Autofac est configuré afin d'être le plus puissant possible. Par exemple, la recherche des constructeurs cherche également ceux qui sont privés/protégés/internals, et ceci afin de ne pas se retrouver bloqué pour mettre en place les bonnes portées sur vos objets.

1.24 DAL avec EF Core

Entity Framework Core permet d'accéder aux données configurée à l'aide de la couche DAL de CQELight. Pour pouvoir l'utiliser, il est nécessaire de créer un contexte propre à votre couche de données qui hérite de `CQELight.DAL.EFCore.BaseDbContext` :

```
public class MyDbContext : CQELight.DAL.EFCore.BaseDbContext
{
    public MyDbContext(DbContextOptions<MyDbContext> options)
        : base(options)
    {
    }
}
```

Avertissement : Pensez à créer cette classe dans le même projet que vos modèles qui sont mappés à l'aide des attributs de CQELight, car une étape de configuration automatique est faite dans la classe BaseDbContext qui recherche automatiquement tous les modèles de son projet pour les mapper dans le moteur Entity Framework Core.

A partir de cette étape, vous avez accès à tous vos objets à l'aide du context préalablement défini. Vous pouvez donc utiliser toutes les fonctionnalités d'Entity Framework Core. Cependant, en faisant cela, vous n'utilisez pas les objets repository (`IDataReaderRepository`, `IDataUpdateRepository` et `IDatabaseRepository`), ce qui vous oblige à utiliser directement les API EF Core partout dans le code.

Bien que cela soit fonctionnel, ce n'est pas optimal, car cela lie très fortement votre code d'accès aux données à Entity Framework Core, vous empéchant ainsi d'utiliser autre chose (MongoDb, NHibernate, . . .). Il est donc conseillé d'utiliser directement les objets repository dans votre code métier.

Avertissement : Ceci nécessite de mettre en place un plugin IoC pour profiter de l'injection de dépendances automatiquement.

L'utilisation de la méthode d'extension du Bootstrapper vous permettra de réaliser cette opération d'enregistrement sans aucun effort sous deux formes :

— Soit en référençant une instance de contexte unique à l'ensemble de votre application (attention, beaucoup de risques de problèmes avec des accès concurrents)
— Soit en référençant des options de contexte pour votre application (option recommandé, le système gère et crée un contexte quand il en a besoin)

De plus, certaines options supplémentaires peuvent être fournies pour déterminer le comportement global de l'accès aux données par le moteur EF Core :

— `DisableLogicalDeletion` : désactive de façon globale la suppression logique (pour n'utiliser que la suppression physique). Cela évite de devoir préciser le flag à chaque suppression.

Ces options sont rajoutées après la configuration des contextes.

```
//With global unique context
new Bootstrapper()
    .UseEFCoreAsMainRepository(new MyDbContext(myDbOptions));

//With global options for all context in all assemblies
new Bootstrapper()
    .UseEFCoreAsMainRepository(myDbOptions);

//With options
new Bootstrapper()
    .UseEFCoreAsMainRepository(myDbOptions, new EFCoreOptions {␣
↪DisableLogicalDeletion = true });
```

A la suite de cette opération, chaque repository qui sera injecté dans votre code utilisera la couche d'accès aux données EF Core pour effectuer ses opérations et verra injecter un `EFRepository<T>` (ou votre sous-instance définie et enregistrée dans le container par vos soins).

Note : Comme précisé dans la page sur l'accès aux données, il est toujours plus intéressant d'utiliser vos héritages de repository afin d'avoir la pleine main sur ce que vous voulez faire. Ici, il s'agirait d'avoir un héritage de `EFRepository` par modèle afin d'avoir des fonctions plus fines. Vous serez de fait assuré d'avoir le contexte qui vous est nécessaire injecté en paramètre de votre repository, si vous enregistrez ce dernier également dans le container IoC.

1.25 Bus In-Memory

Le bus In-Memory fais partie du coeur du système de CQELight. C'est lui qui fait transiter la totalité des messages (événements et commandes) d'un point à un autre. C'est aussi lui qui se place en sortie des autres bus afin d'envoyer les informations dans le système courant. Bien qu'il soit disponible sous forme de package, il y a fort à parier qu'il soit un élement central et indispensable de votre futur applicatif. Le bus In-Memory fonctionne sur un principe simple : il scanne les assemblies de votre projet pour y trouver l'ensemble des handlers de destination de l'information qu'il a à envoyer.

Comme une grande partie de nos composants, le bus In-Memory est hautement configurable pour faire face aux besoins que vous pourriez avoir. Afin de l'utiliser, il suffit simplement d'appeler la méthode d'extension `UseInMemoryEventBus` et/ou `UseInMemoryCommandBus`. Les deux méthodes ont la même logique d'appel : soit on fourni un instance de la classe d'options, soit on configure le composant à l'aide d'une lambda. Pour finir, on

peut également préciser une collection de noms d'assemblies à exclure pour la recherches d'handlers afin d'améliorer les performances.

A noter cependant que ces paramètres de personnalisation sont totalement optionnels et l'on peut se contenter des appels les plus simples possibles que sont `myBootstrapper.UseInMemoryEventBus()` et `myBootstrapper.UserInMemoryCommandBus()`.

```
//without options
myBootstrapper.UseInMemoryEventBus();
//with options
myBootstrapper.UseInMemoryEventBus(opt => { opt.NbRetries = 3; });
```

Etant donné que l'envoi se fait en mémoire, il peut être nécessaire d'avoir envie d'ordonner ou de gérer la priorité de réception. A cet effet, un attribut est disponible : `HandlerPriority`. Il suffit de le mettre au dessus de la déclaration d'une classe, avec la valeur désirée, pour que cet handler soit placé comme il se doit dans la liste des envois.

..warning : : Il n'est pas possible de garantir la priorité d'un handler par rapport à un autre si les deux possède la même valeur dans l'attribut. De même, cette valeur n'est pas considérée en cas de gestion en parallèle.

De la même manière, il peut être parfois nécessaire, dans votre solution, de s'assurer qu'un handler soit exécuté avec succès avant de passer aux autres. Ces handlers sont qualifiés de « critiques », et doivent donc être marqués avec l'attribut CriticalHandler.

..warning : : Cette notion de criticité n'est valide que si l'appel des handlers est fait de façon procédurale, c'est à dire un après l'autre. Cette valeur n'est pas considérée en cas de gestion en parallèle.

Lorsque votre projet commencera a devenir conséquent, il y a fort à parier que vous n'aurez pas ou plus la vision globale de la configuration. Afin de palier à cela, le bootstrapping renvoie une collection de notification vous aidant à voir s'il y a des failles dans votre configuration. Ces notifications sont soumises au flags "Strict" et "Optimal" du bootstrapper. La totalité des notifications sont des warning, il est donc nécessaire d'être vigilant et de surveiller le retour de la fonction Bootstrapp.

Plusieurs options sont à votre portée pour configurer le bus In-Memory d'événements :

- `WaitingTimeMilliseconds` : temps d'attente entre deux essais de livraison d'événement, en cas d'échec lors de la première tenative.
- `NbRetries` : nombre d'essai lorsqu'une livraison d'événement n'a pas pu se dérouler comme prévu.
- `OnFailedDelivery` : callback invoqué avec l'événement et le contexte associé lorsque ce dernier n'a pas pu arriver convenablement à destination.
- `IfClauses` : conditions particulières indiquant si l'événement doit être envoyé dans le système ou non.
- `ParallelHandling` : flag par type d'événement autorisant le système à gérer le handling parallèle d'une instance du type d'événement configuré. Attention : incompatible avec les attibruts de priorité et de criticité.
- `ParallelDispatch` : flag par type d'événement autorisant le système à propager l'événement en parlalèle dans le système.

Il y a également quelques options pour la configuration du bus In-Memory de commandes :

- `OnNoHandlerFounds` : callback invoqué avec la commande et le contexte associé en cas d'absence de handler pour la traiter.
- `IfClauses` : conditions particulières indiquant si la commande doit être envoyée dans le système ou non.
- `CommandAllowMultipleHandlers` : configuration pour un type de commande particulier afin d'indiquer plusieurs handlers sont autorisés ou non.

1.26 Event sourcing avec EF Core

Il est possible d'utiliser Entity Framework Core, et donc de profiter de la flexibilité de provider qu'il intègre, pour faire un système event-sourcé. Les événements seront persistés dans une base de données relationnelle. Bien que cela ne soit pas sa fonctionnalité initiale, cela permet d'avoir un système fonctionnel rapidement et simplement.

Etant donné que les bases de données relationnelle utilisent un schéma pour la persistance des données, il est nécessaire de créer une migration EF Core dans votre projet. Il faut en premier lieu définir dans le projet pour qu'il sache comment configurer le contexte :

```
// With SQLite
public class EventStoreDbContextCreator : IDesignTimeDbContextFactory
↪<EventStoreDbContext>
{
    public EventStoreDbContext CreateDbContext(string[] args)
    {
        return new EventStoreDbContext(new DbContextOptionsBuilder
↪<EventStoreDbContext>()
                    .UseSqlite("FileName=events.db", opts => opts.
↪MigrationsAssembly(typeof(EventStoreDbContextCreator).Assembly.GetName().Name))
                    .Options, SnapshotEventsArchiveBehavior.Delete);
    }
}
```

Une fois ceci fait, il est nécessaire de créer une migration EF Core. Pour ce faire, il faut lancer la commande suivante sur votre projet exécutable :

```
// Dotnet CLI
dotnet ef migrations add EventStoreMigration -c EventStoreDbContext
// VS
Add-Migration EventStoreMigration -c EventStoreDbContext
```

La migration est ajoutée à votre projet. La dernière étape pour utiliser EF Core comme EventStore et de le déclarer dans le bootstrapper :

```
// With SQLite
new Bootstrapper()
            .UseEFCoreAsEventStore(
            new CQELight.EventStore.EFCore.EFEventStoreOptions(
                c => c.UseSqlite("FileName=events.db", opts => opts.
↪MigrationsAssembly(typeof(Program).Assembly.GetName().Name)),
                archiveBehavior: CQELight.EventStore.SnapshotEventsArchiveBehavior.
↪Delete))
            .Bootstrapp();
```

Le code de configuration du le `DbContextionOptionsBuilder` peut être mutualisé afin d'être écrit une seule fois. Le contexte n'est pas ajouté dans le container IoC d'un projet Asp.Net Core, comme on pourrait le faire avec `services.AddDbContext`. Ceci est du au fait qu'il est déconseillé d'utiliser directement le contexte pour accéder à l'event-store, et qu'il est recommandé d'utiliser les interfaces `IAggregateEventStore` et `IEventStore` (ou `IReadEventStore` et `IWriteEventStore`), car beaucoup de règles de gestion sont implémentées dedans et ne sont pas disponibles au niveau du contexte EF. Utiliser le contexte EF directement pourrait compromettre l'intégrité de votre EventStore, surtout en écriture.

1.26.1 Spécificités

Le provider EF Core dispose de certaines spécificités permettant d'optimiser le traitement avec la base relationnelle. La classe `EFEventStoreOptions` permet de préciser chacun de ces spécificités.

— `SnapshotBehaviorProvider` et `ArchiveBehavior` permettent de précisier le mode de fonctionnement du moteur de snapshot. Pour plus de renseignements sur la notion de snapshot, voir la page sur l'event sourcing.
— `DbContextOptions` définit le mode d'accès à la base principale des événements

— `ArchiveDbContextOptions` définit le mode d'accès à la base d'archive des événements. Note : cette propriété est obligatoire si la valeur du membre ArchiveBehavior est définie à `StoreToNewDatabase`
— `BufferInfo` permet de définir le comportement du tampon utilisé pour optimiser les requêtes vers le SGDB.

La notion de buffer a été ajoutée pour éviter de faire trop d'appels à la base dans le cadre d'un système très sollicité par l'envoi d'événements unitaires. Par exemple, si on imagine un système qui propage un événement toutes les 200 millisecondes, on risque de se retrouver avec utilisation intensive d'EF Core et du système transactionnel qui va ralentir notre event-store. Les membres suivants sont disponibles :

— `UseBuffer` : flag d'activation
— `AbsoluteTimeOut` : Timeout absolu à partir duquel les événements doivent être persistés obligatoirement
— `SlidingTimeOut` : Timeout glissant permettant de définir une durée de persistance à partir de laquelle les événements doivent être persistés s'il n'y en a pas de nouveaux

Deux configurations sont disponibles par le biais de variables statiques globales : `BufferInfo.Enabled` et `BufferInfo.Disabled`. L'utilisation de la valeur Enabled utilisera une valeur de 10 secondes en timeout absolu et 2 secondes en timeout glissant.

Il faut être vigilant avec l'utilisation du buffer, car s'il est améliore effectivement les performances sur les systèmes qui sont souvent sollicités en terme de propagation d'événements, il va ralentir un système qui ne propage pas énormément d'event. Pour savoir s'il vous faut l'utiliser, faites une statistique du temps moyen de propagation d'événement dans votre système et voyez si ce temps moyen est inférieur à 2 secondes. Si oui, considérez l'utilisation du buffer. Si non, ne l'activez pas (par défaut).