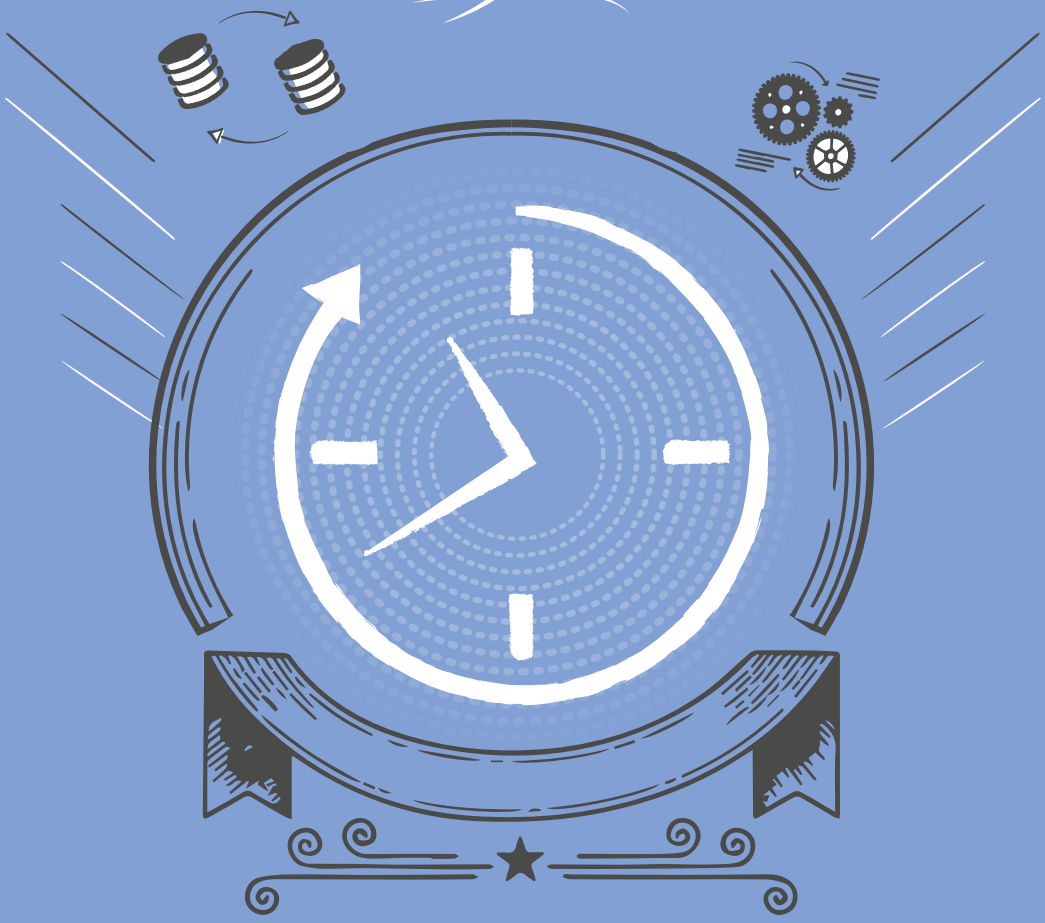
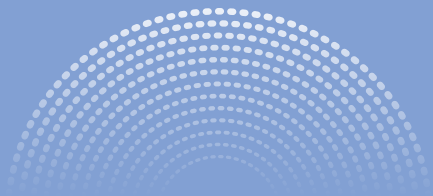
SOAT



CQRS & Event Sourcing

LIVRE BLANC



SOAT en quelques mots

Fondé par Michel AZRIA et David-Eric LEVY, SOAT est un cabinet de conseil IT qui entraîne depuis 17 ans ses collaborateurs, dans une aventure où se conjuguent expertise, bonne humeur et agilité.

Au quotidien, ce sont plus de 360 collaborateurs de talent qui accompagnent nos clients sur des problématiques technologiques et organisationnelles complexes : développement d'applications spécifiques, performance et qualité du code, déploiement continu, choix technologiques, conception d'architectures dynamiques et évolutives, et transformation agile à grande échelle.

Pour garantir à nos clients des prestations de haute qualité et une expertise toujours à la pointe, nous avons placé la capitalisation de nos savoir-faire au cœur de notre stratégie.

Conférences, livres blancs, avis d'expert, blog, communautés techniques et de pratiques… Nous favorisons le partage des connaissances pour faire continuellement progresser nos consultants et apporter toujours plus de valeur ajoutée à nos clients. Cette démarche de partage et de capitalisation ancrée dans les gènes de SOAT participe à la diffusion des savoirs.

Allier savoir-faire et savoir-être reste notre défi au quotidien pour rendre notre organisation toujours plus performante et épanouissante.

Table des matières

Introduction

"We are not code writers, we're problem solvers."

Michael Feathers*

Des motifs architecturaux pour répondre aux enjeux de la mise à l'échelle des applications.

Elle avait l'air plutôt simple, cette demande d'évolution. Il suffisait de rajouter une table dans la base de données, ainsi qu'une clé étrangère, d'enrichir quelques requêtes avec une jointure supplémentaire, de modifier 4-5 services applicatifs pour remonter la nouvelle donnée, et de compléter un ou deux écrans d'une textbox… Le quotidien du développeur, en somme. Les estimations semblaient généreuses, le socle technique éprouvé, quoiqu'encombré avec les années. Alors pourquoi les temps de chargement ont tant augmenté ? Pourquoi les régressions se sont multipliées ? Pourquoi la qualimétrie s'est affolée ? Et pourquoi les délais ont dérapé ?

Une situation loin d'être inconnue ? En effet ! L'application vient d'éprouver subitement les limites de sa conception CRUD (Create/Read/Update/Delete) et des implications qu'elle entraîne tant en termes de gestion de la modélisation du métier complexe qu'en termes de scalabilité. Cette modélisation, pourtant simple à mettre en place et à transmettre, ne serait-elle pas devenue un frein à l'agilité globale de l'application, aussi bien du point de vue métier que technique ? Si une évolution aussi "simple" prend autant de temps et d'efforts, comment s'adapter aux changements du marché, toujours plus rapides ?

Parmi les axes de travail qui vont permettre de rétablir une délivrabilité optimale, nous allons traiter dans ce livre de deux motifs architecturaux qui permettent d'améliorer les performances et l'évolutivité des applications. Le premier de ces motifs d'architecture logicielle a été formalisé en 2010 par Greg Young**. Il s'agit de CQRS (Command and Query Responsibility Segregation), un motif architectural visant à séparer les lectures des écritures de données.
Le second est l'Event Sourcing. Formalisé dans les années 2000, notamment par Martin Fowler*** et Greg Young (encore lui !), mais théorisé dès les années 60 (*Situations, Actions, and Casual Laws, John McCarthy, 1963*) il s'agit d'un pattern de persistance de la donnée basé non pas sur les états successifs des concepts métiers dans un système, mais sur les faits qui se sont déroulés dans le système.

Les personnes les plus aptes à exprimer les besoins du métier de manière explicite et précise, sont les experts du domaine, qu'ils soient Business Analysts ou utilisateurs finaux. Pour appliquer le plus pertinemment possible les patterns d'architecture CQRS et Event Sourcing, les développeurs et architectes devront donc être en mesure de collaborer directement avec eux.

* Extrait de « Working effectively with legacy code » - 2004 ** Greg Young : Auteur, Conférencier et Formateur sur DDD, CQRS & Event Sourcing *** Martin Fowler : Craftsman & Auteur du livre « Refactoring »

I PARTIE

Command and Query Responsibility Segregation

Mon modèle de données me ralentit !

Dans les architectures applicatives traditionnelles, les mêmes objets sont utilisés à la fois pour la lecture et l'écriture. Il est donc cohérent d'avoir une source de données unique pour ces deux tâches. Une base de données relationnelle est la technologie de persistance la plus naturelle et la plus répandue, notamment grâce à la maturité acquise par les technologies SQL en termes de fiabilité et de performance.

Pour autant, la promotion de la troisième forme normale dans la modélisation des données induit des effets négatifs. Le modèle métier tend à ne devenir qu'un reflet du modèle de données, débarrassé de son comportement, devenant ainsi anémique (entité ne contenant que des propriétés et n'implémentant aucun comportement). Obtenir des données depuis la base nécessite dès lors des jointures complexes et coûteuses.

Le motif architectural CQRS propose d'adresser ces deux problématiques en séparant complètement les responsabilités de lecture (Query) et d'écriture (Command) en deux modèles de données distincts. Il permet en outre d'adresser les déséquilibres de volumétrie entre consultation des données et évolution de l'état de l'application.

Il est important de garder à l'esprit qu'une commande (Command) change l'état du système et une demande (Query) ne change pas l'état du système mais retourne un objet. Si ces deux responsabilités ne sont pas distinctes, cela signifie que le pattern CQS (Command Query Separation) n'est pas respecté.

Prenons l'exemple d'un site de vente en ligne. La consultation du catalogue est une opération effectuée de manière bien plus fréquente que la validation d'un panier. Cette consultation n'implique pas de logique métier à proprement parler, alors que la confirmation de commande va déclencher une vérification de la disponibilité des articles dans le stock, la mise à jour dudit stock, la préparation d'un envoi (cf. schéma Validation de panier).

Avec une base relationnelle, la validation d'un panier implique l'ouverture d'une transaction technique, qui va opérer un ensemble de vérifications et de modifications durant lesquelles certaines données seront verrouillées (comme la quantité d'articles disponibles) pour être finalement mises à jour. Le temps de cette transaction, les performances en lecture sont impactées.

Avec un modèle anémique, la validation du panier prendra place, non pas au sein du modèle métier, mais dans un service qui aura la charge de mettre en place et d'exécuter la transaction, ce qui va nuire à l'encapsulation et donc à la maintenabilité du modèle métier.

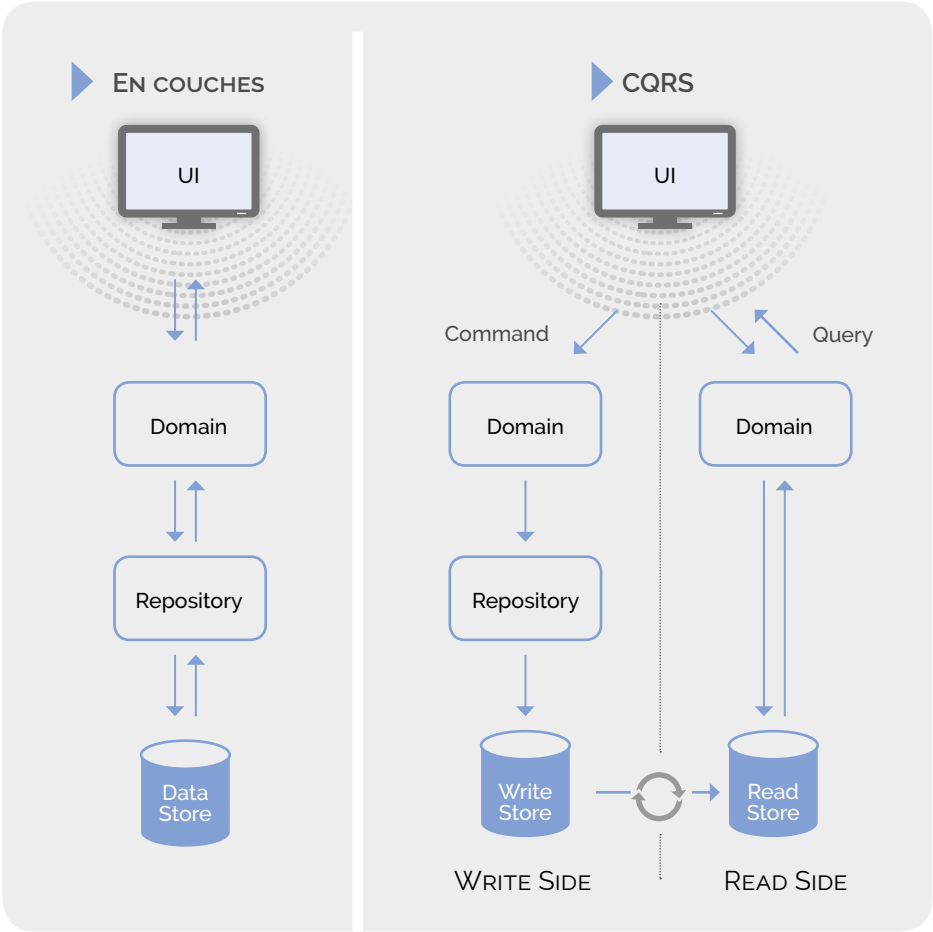
Ce modèle applicatif (script transactionnel) pénalise d'une part la lecture des données au profit de leur écriture, et d'autre part, sa maintenabilité.

VALIDATION DE PANIER

Le modèle CQRS

Deux modèles sinon rien : Cassons la base !

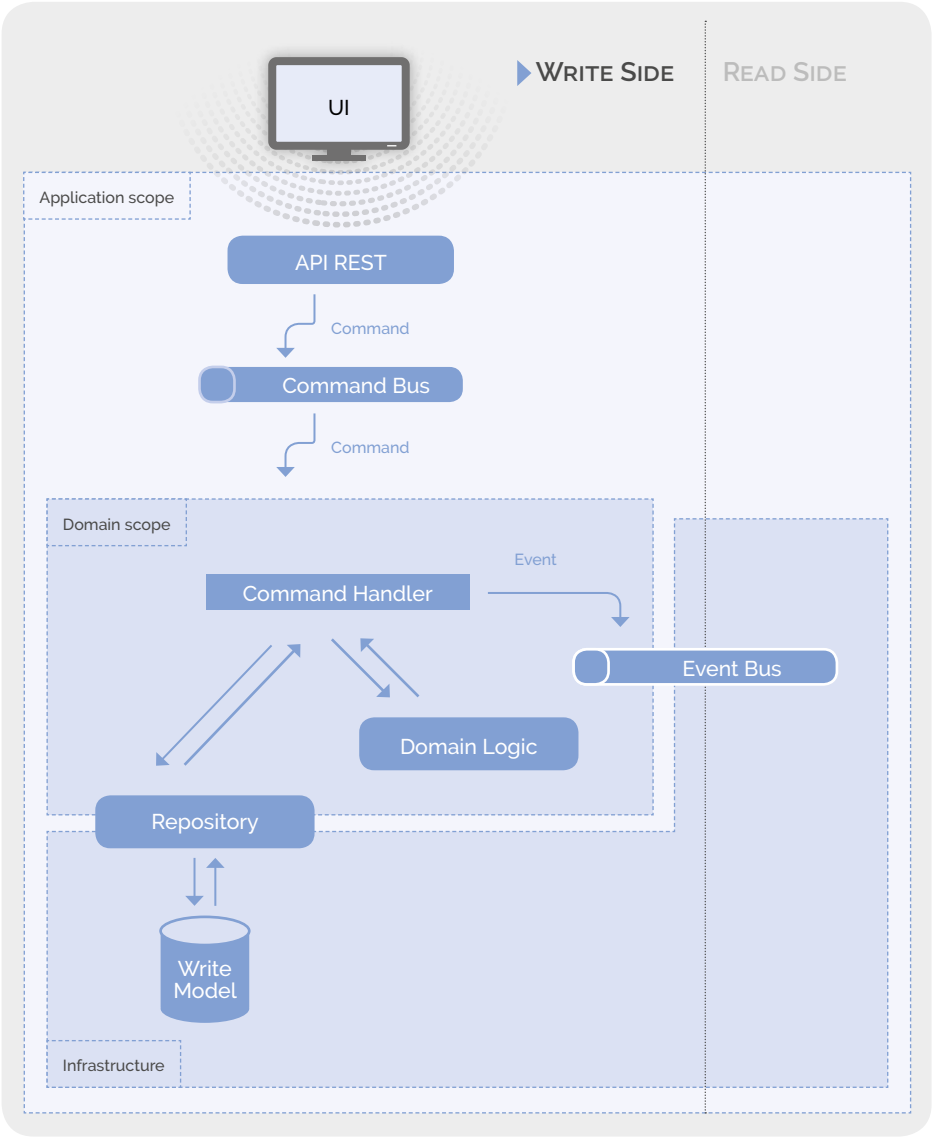
Depuis l'avènement des bases de données relationnelles et de SQL, les applications se reposent énormément sur cette technologie. Si les moteurs SQL sont devenus de plus en plus performants et capables de traiter de très grandes quantités de données dans des temps toujours plus courts, ils incitent toutefois à créer des modèles relationnels qui tendent vers la forme la plus normalisée possible, afin de garantir l'intégrité de la donnée vis-à-vis des écritures. Cette structuration de la donnée est optimisée pour l'écriture. Séparer la persistance en deux parties avec des responsabilités spécifiques permet d'optimiser chaque partie du modèle indépendamment (indexation, clustering, sharding, etc...)



Write Side

Composition du Write Side :

Dans une architecture CQRS, le Write Side est constitué de plusieurs composants distincts :



1. Les Command Handlers

Les Command Handlers sont chargés de traiter les commandes entrantes, et de les exercer sur le Domain Model. Ils correspondent aux implémentations des Applications Services*, les modules applicatifs prenant en charge les cas d'utilisation métiers au sein de l'application. Dans le cas d'une application cliente unique, ces Command Handlers peuvent être déportés au plus près de la réception des commandes. Par exemple, dans les applications mettant en place les motifs MVC ou MVVM, les Controllers ou les ViewModels** peuvent endosser le rôle de Command Handler.

⚠ Le 'Model' de MVC et MVVM ne désigne en aucun cas le Domain Model. Il s'agit seulement de la définition de la communication avec les clients.

Dans les projets de plus grande envergure, cependant, ces Controllers ou ViewModels auront la responsabilité du contrôle de surface des stimuli externes et la construction de Command qui seront dispatchés aux différents Command Handlers avec un Command Pattern tel que décrit dans le livre du GOF (Gang of Four, acronyme utilisé pour désigner le groupe des quatre auteurs : Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides) : Design Patterns : Elements of Reusable OO Software ou un Command Bus.

⚠ Le Command Pattern du GOF ne représente pas le même motif qu'une Command dans CQRS. Les Command CQRS sont de simples messages, des DTO (Data Transfer Objects) non modifiables, qui peuvent être pris en charge au moyen d'un Command pattern du GOF.

Une commande ne peut être prise en charge que par un seul Command Handler. Dans le cas contraire, il est fort probable que les frontières transactionnelles soient mal conçues/défectueuses. Il faudra alors soit revoir les frontières transactionnelles de la Commande, soit mettre en place un Process Manager, c'est à dire un Event Handler Stateful, qui en réponse à un Domain Event produira une nouvelle commande relative à un autre objet métier (au sens d'agrégat) pouvant entraîner un traitement complexe.

Notre avis : ***N'hésitez pas à enrober les commandes et les events dans une structure portant des métadatas afin de pouvoir suivre les conséquences d'une commande dans le système.***

2. Le Domain Model

Il s'agit de la modélisation du domaine que l'application adresse. C'est dans le Domain Model que réside l'implémentation de l'ensemble des règles métiers, et des éléments qui ont un sens pour les experts du domaine. Ce modèle décrit et implémente les Domain Events, les Commands, les Command Handlers, les contrats nécessaires pour que ce modèle puisse interagir avec les éléments extérieurs, comme les interfaces décrivant les fonctionnalités du Write Store.
Idéalement, le Domain Model devra s'abstraire de toute considération technique, son unique responsabilité étant de réaliser les cas d'usage du métier.

Notre avis : ***Nous vous incitons fortement à utiliser une approche DDD pour implémenter ce Domain Model. DDD vous permettra d'obtenir un modèle suffisamment encapsulé pour qu'il puisse évoluer sereinement dans le temps.***

* NB 1 : une application service correspond à un seul et unique scénario métier ou use case
** NB 2 : il s'agit du « Open Host »

3. L'Event Bus et les Events Handlers

Une fois qu'une commande a été exercée contre le Domain Model, ce dernier va publier un Domain Event sur l'Event Bus, auquel seront abonnés des Events Handlers du Read Side et du Write Side. En fonction de la taille du système, de la concurrence qu'il abrite, de sa distribution (ou non), différentes techniques peuvent être mises en place. Dans les systèmes les plus simples (In-Proc CQRS), on peut parfaitement se reposer sur un simple dispatcher, voire un Observer pattern. Dans les systèmes plus complexes, notamment quand ils sont distribués, un mécanisme de file de messages sera avantageux.
Au sein du Write Side, le rôle principal des Event Handlers est de gérer des processus métiers. Ils deviennent alors des machines à états activant leurs transitions lors de la réception d'un Domain Event et publiant une commande lors des Transitions.

4. Le Write Store

Dans le Write Side, le write store a deux responsabilités bien distinctes.
La plus évidente est de persister les objets du domaine et les modifications qui y sont apportées, une responsabilité de pure écriture.
La seconde est un peu moins évidente, car il s'agit de lire l'état courant des objets à manipuler afin de les réhydrater dans la partie applicative, pour permettre au Domain Model de réaliser les cas d'utilisation métiers. Les modifications d'un objet métier* devant être transactionnelles, ces données ne peuvent parvenir du Read Side, au risque de remettre en cause l'intégrité des données dans le Write Side.
En fonction de la complexité métier à gérer et des contraintes techniques, le write store peut se baser sur différentes infrastructures.

La plus classiquement utilisée est une base de données relationnelle, qui conviendra à cet usage pour peu que la structure des données soit optimisée, et non pas un simple reflet du modèle métier. D'autres technologies peuvent être mises en place, comme un magasin d'événements, comme décrit plus en détails dans la partie Event Sourcing (p.14)

5. Le Command Bus

Dans les systèmes de grande envergure, il est possible de distribuer le Write Side entre plusieurs machines, voire entre plusieurs applications sur des instances dédiées au traitement de la logique métier. Il arrive aussi que certaines commandes aient des délais de traitement particulièrement longs, notamment quand la charge augmente. Afin de s'assurer que l'ensemble des commandes émises par le système soit prises en compte et traitées, il devient nécessaire d'introduire un nouveau composant dans le Write Side, le Command Bus. Ce dernier a pour objectif de garantir la mise en file des commandes ainsi que leur routage vers les Command Handlers appropriés, et peut offrir un mécanisme de résilience pour des reprises sur erreur quand une application traitant un ensemble de commandes, redémarre suite à un plantage. Cependant, dans des applications faiblement concurrentes où l'ensemble des composants de la chaîne CQRS résident dans le même process (ou In-Proc CQRS), ce composant sera avantageusement remplacé par un simple dispatcher de commande.

Quand un système s'enrichit, laisser aux controllers le rôle de Command Handler mène rapidement à des Fat Controllers, qui deviennent difficile à maintenir. L'idée est donc d'ôter au controller le rôle du Command Handler.

*NB : nous parlons ici d'objet métier riche, un agrégat au sens DDD

Read Side

Projections

Dans un système classique de grande ampleur, exploitant une base de données relationnelle de plusieurs centaines de tables, et qui utilise un modèle de données unique pour les écritures et les lectures, les données sont généralement architecturées pour garantir leur intégrité et leur cohérence.

La *projection* des données peut alors rapidement devenir complexe, nécessitant une grande quantité de jointures.
L'optimisation la plus évidente consiste à concevoir une indexation efficace pour accélérer l'extraction des données, mais cela a un impact sur les performances en écriture.
Dans certains cas extrêmes, les projections nécessitent la jointure de plusieurs dizaines de tables, comme le montre le diagramme ci-contre. Ce type de requête, en raison du temps d'exécution important, ne garantit pas une cohérence absolue des résultats.

Le second type d'optimisation qu'il est possible de mettre en place sera la mise à plat des données pour cette requête spécifique. Cette mise à plat implique une duplication des données afin de les concentrer dans une structure optimisée pour la lecture. Ce processus se nomme "dénormalisation".

La dénormalisation de la donnée impose une synchronisation entre la donnée primitive et la donnée dérivée.
Avec CQRS, cette synchronisation est prise en charge par les dénormalisateurs, qui réagissent aux Domain Events émis par le Domain Model dans le Write Side.

Le travail de ces dénormalisateurs est généralement activé par une réponse aux stimuli d'un ou plusieurs Domain Events au moyen d'event handlers présents dans le Read Side. Les différentes dénormalisations présentes dans le système s'apparentent alors à des vues matérialisées, nommées projections.

La pratique recommande d'utiliser un dénormalisateur par type de projection. Chaque projection est construite avec un objectif spécifique. Dans l'exemple du site e-commerce, lors du paiement de la commande, un domain event est émis, et sera consommé par plusieurs dénormalisateurs :

- Gestion du stock
- Statistiques globales d'achats du jour
- Historique des commandes du client

Chacune de ces projections sert un but spécifique, identifié en amont, et contient exactement les données nécessaires, de sorte que l'accès aux données soit aisé et efficient.

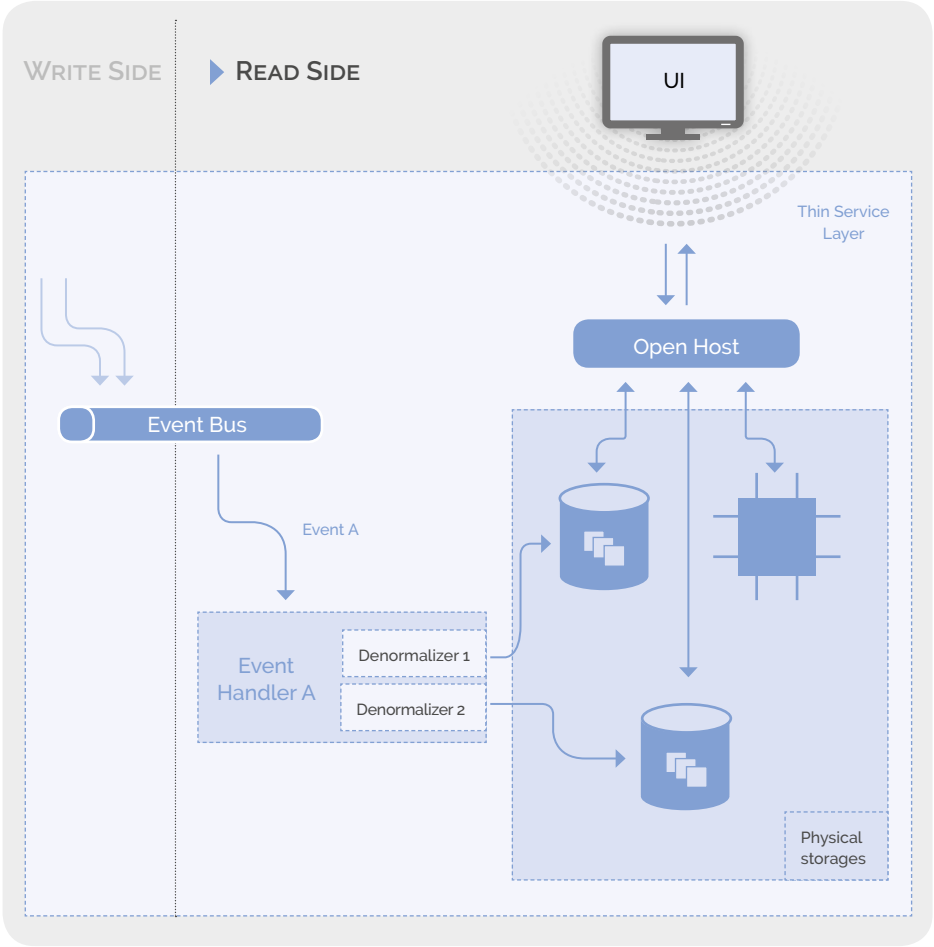
- Aisé, car la représentation des données dans une projection du Read Side est lue avec uniquement l'identifiant du concept en cours d'accès (l'identifiant du client pour l'historique des achats, le jour/mois voulu pour les statistiques de vente, le Stock Keeping Unit, SKU de l'article dans le stock)
- Efficient, car il n'y a pas de jointure à effectuer pour obtenir l'ensemble des données nécessaires, ces dernières se trouvant à plat dans une structure spécifique, par exemple, un document dans une base documentaire.

Points d'attention pour les projections

- La Structure des projections est immuable, si le besoin d'une projection change, c'est qu'une nouvelle projection est nécessaire.
- Sans Event-Sourcing, créer de nouvelles projections demande une reprise de données pour intégrer l'existant.
- Si une projection n'est plus utilisée, la pratique admise est de la supprimer, car les coûts de maintenance et de performance peuvent s'avérer élevés.

Composition du Read Side

Le Read Side dispose de 2 connectivités ; l'une entièrement dédiée à la récupération de données, et l'autre, sur un ou plusieurs système de messages, chargée de mettre à jour les données depuis les Domain Events (aussi appelés Integration Events dans ce rôle spécifique). Le Read Side ne réalise aucune action métier. Ses deux seules et uniques responsabilités se limitent à dénormaliser et structurer les données en provenance du Write Side, et à offrir une consultation de ces données.



Event Bus & Integration Events

Il s'agit du point d'entrée d'écriture des données dans le Read Side. Le Read Side est connecté au Write Side de manière découplée via l'Event Bus. L'Event Bus étant le rôle affecté au composant chargé de faire circuler les Domain Events dans le système. La technologie portant ce dernier doit correspondre aux contraintes métiers en termes de résilience et de résistance aux pannes, ainsi qu'aux contraintes définies dans l'architecture d'entreprise en termes de distributivité. Les Domain Events circulant dans ce bus et consommés par le Read Side sont parfois nommés des Integration Events.

Event Handlers

Dans le Read Side, les Event Handlers ont pour seule responsabilité de consommer les Domain Events intéressant le Read Side, afin de les transmettre aux dénormalisateurs. Contrairement aux Event Handlers du Write Side, ils ne déclenchent pas d'action et ne produisent donc jamais de nouvelles commandes.

Dénormalisateurs (denormalizers)

Quand un Event Handler du Read Side consomme un Domain Event, il le transmet à un ou plusieurs dénormalisateurs. Leur rôle est de préparer les données afin qu'elles puissent être lues le plus rapidement possible. Il s'agit donc d'algorithmes d'aplatissement de données. Une fois aplaties et mises en forme, les données sont persistées par le dénormalisateur.
Un dénormalisateur a la charge d'alimenter une unique projection, et sera donc appelé par chaque Event Handler du Read Side qui requiert la mise à jour du read model.

Persistance(s)

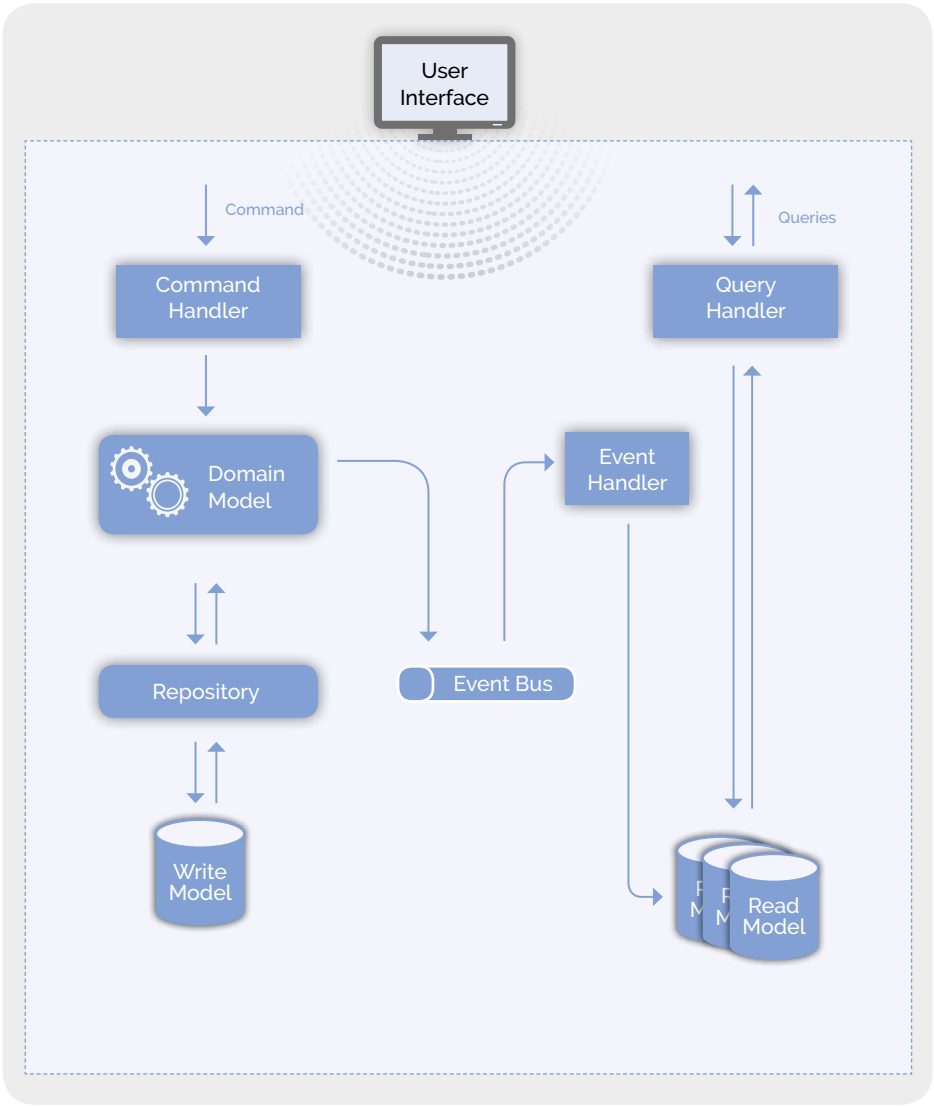
La persistance des données dans le Read Side doit reposer sur les technologies les plus adaptées aux contraintes de chacune des projections. Il est complètement envisageable d'opter pour plusieurs technologies différentes, le but étant ici d'obtenir la meilleure performance possible en requête.
En l'espèce, certaines projections reposent sur une structure purement documentaire, c'est-à-dire un couple (Identifiant, Données à plat), et sont consommées en tant que telles. Dans ce cas, une base de données documentaire est très efficace, à la fois en lecture et en écriture, d'autant plus si le mode "append-only" de version du document est privilégié.

Certaines projections n'ont pas à être persistées plus de quelques heures, comme une session de navigation sur un site e-commerce. Dès lors, une persistance dans un cache mémoire est parfaitement adaptée. D'autres projections, encore, se doivent d'offrir des mécanismes de requêtage poussés. Dans ce cas, l'utilisation de bases de données relationnelles ou de bases de données orientées graphes, est tout à fait cohérente. Un cas d'usage typique de cette technologie dans le e-commerce est la réalisation d'un moteur de cross-selling.

Service Layer

Le Read Side est consommable depuis une fine couche de service, responsable d'extraire les données depuis la persistance du Read Side, et de les rendre aux appelants. Cette couche doit être la plus simple et la plus fine possible, et ne pas contenir de logique métier.
L'utilisation de ce service layer ne se limite pas seulement aux divers clients de l'application, le Write Side peut aussi en tirer parti. Comme les données y sont accessibles en lecture de façon très performante, il peut être pertinent de requêter certains sets de données dans les Command Handler avant de les transmettre à la logique métier qui pourra alors effectuer des validations de certaines règles métiers sur ces sets. Il faudra toutefois garder à l'esprit la cohérence finale de ces sets de données.

CQRS : diagramme complet :



Do and Don't

Don't : On n'applique pas le pattern partout !

Ce motif architectural est efficace, mais il ajoute une complexité technique à ne pas négliger. Dans le cas d'applications ou de sous-domaines très simples, sans enjeux de performance, CQRS ne sera pas forcément adéquat.

Don't : CQRS sur les cas d'usage simples

Certains sous-systèmes, certaines applications ne comportent aucune logique métier. Une application de gestion d'un référentiel de zones géographiques, par exemple, chargée de lier des villes à des pays, et des pays à des continents, ne nécessite pas de cas d'usage complexes. Dans ce type d'application, CQRS est sur-dimensionné.

Do : Parfois, CRUD suffit

Lorsque la raison d'être d'un applicatif est de stocker et de restituer de la donnée sans aucune logique ni aucun comportement particulier implémenté dans les modèles, CQRS n'est pas nécessairement la solution à mettre en place. En effet, la complexité engendrée par cette architecture dépasserait largement la complexité intrinsèque de l'applicatif. Dans ce type de cas, une application CRUD suffit amplement.

Do : On encapsule notre domaine

CQRS aide à se focaliser sur les cas d'usage métiers que l'application adresse. Dès lors, ce motif architectural se mariera idéalement avec DDD (Domain Driven Design) ou la conception pilotée, en français. Le command side ou write model exercera alors les cas d'utilisation contre les agrégats (des grappes d'objets ultra cohérents) du modèle métier. Les Command Handlers auront un rôle d'application service (contient simplement le scénario d'un cas d'utilisation), et le modèle d'écriture deviendra le Domain Model.
Le Domain Model étant riche, ce dernier encapsulera la logique métier ainsi que sa complexité, facilitant par là l'implémentation de nouveaux cas d'usage.

⚠ les applications clientes peuvent être de natures variées, et avoir chacune leur protocole d'application. Il est préférable d'unifier l'accès à l'application via un protocole unique (HTTP/REST, par exemple) mais ce protocole n'influe pas sur l'architecture CQRS.

II PARTIE

Event Sourcing

Pourquoi se contenter de ne conserver qu'une photographie des données ?

Qu'est-il arrivé aux données ?

Pourquoi un article sur un site d'e-commerce se vend-t-il mal ? Est-il trop cher, pas assez spécifique ou différenciant ? A-t-il été mal marketé ? Pourquoi aucun client ne l'ajoute à son panier ? Ou alors, pourquoi de nombreux clients le retirent avant même de valider leurs achats ?

Si connaître le contenu d'un panier au moment de son paiement est indispensable, connaître l'évolution même de ce panier à toutes les étapes du parcours client va permettre aux directions marketing ou produit, de mieux comprendre les habitudes des clients, leurs hésitations ou freins à l'achat, et d'évaluer plus justement l'efficacité de certaines opérations (promotions, suggestions d'achat, etc.)

L'état d'une chose ou d'un concept, est en fait dérivé des événements relatifs à cette chose, à ce concept, qui se sont produits. Sur un site internet, un panier ne se remplit pas ex-nihilo : un client démarre une session de navigation, ce qui a pour effet d'initialiser son panier. Il ajoute ensuite des articles, en retire certains, en ajoute d'autres… Si au moment de valider sa commande, le panier d'un client contient un certain ensemble d'articles, c'est par l'historique des ajouts et des suppressions que cet ensemble est ce qu'il est. La donnée primitive correspond aux décisions prises par le système, suite à l'expression d'une intention du client. L'ensemble d'articles constituant le panier est le résultat de ces événements.

La dérivation entraînant une perte d'informations, il est donc plus judicieux de considérer la persistance de la donnée primitive plutôt que celle de la donnée dérivée. C'est l'intention première de l'Event Sourcing : focaliser la persistance sur l'origine des données plutôt que sur l'état du système.

Pourquoi se concentrer sur les événements plutôt que sur les commandes ?

Comme précisé dans le chapitre CQRS, les commandes reflètent l'intention des utilisateurs, là où les événements métiers expriment des faits qui se sont produits dans le système. Le métier évoluant, la façon de réagir aux intentions des utilisateurs évolue elle aussi. En revanche, les conséquences des décisions métiers, une fois prises, ne changent pas. Quand une règle de gestion change, elle ne remet pas en cause le fait qu'un événement se soit produit dans le passé.

Concentrer un modèle de persistance autour des événements métiers sera donc bien plus judicieux et pérenne.

En plus des opportunités Business, de nouvelles opportunités IT

Le fait de stocker les données primitives plutôt que les données dérivées génère également de nouvelles opportunités IT à moindre coût. Quand l'ensemble des opérations, des décisions métiers prises par un système ou par un sous-système, servent de socle pour reconstruire l'état dudit système, un audit exhaustif des actions métiers est alors disponible dans le coeur du SI, sans effort supplémentaire.

La conception itérative d'un tel système est elle aussi facilitée, car les spécifications fonctionnelles deviennent plus simple à exprimer :
*"**Etant donné** un objet auquel des événements donnés sont déjà arrivés,*
***quand** il reçoit tel ordre,*
***alors** tel nouvel événement est émis par le système".*
Des tests automatisés très explicites peuvent alors être mis en place, et la quantité de use cases couverts peut augmenter de manière significative.

De plus, le débogage de l'application peut être grandement facilité, notamment pour les investigations en production, car l'état et l'historique d'un objet posant problème peuvent être facilement rapatriés et exécutés dans un environnement dédié, comme le poste de travail d'un développeur, par exemple.

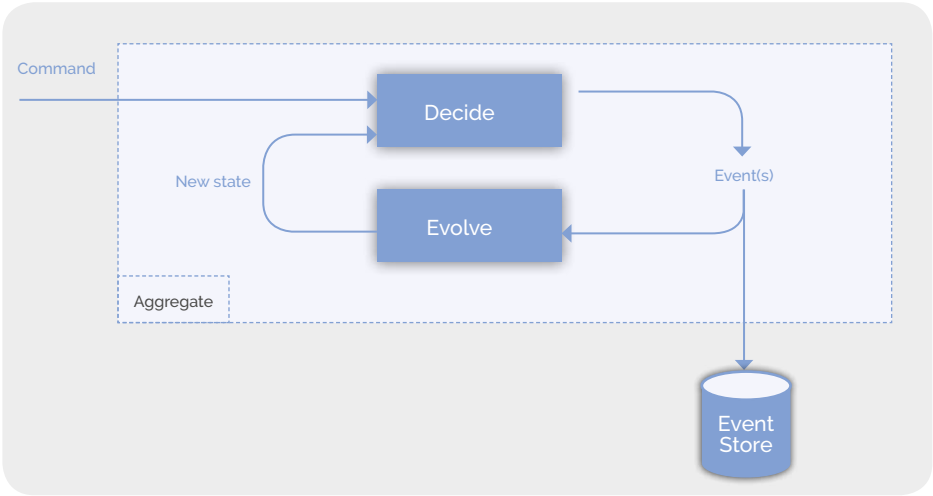
De nouvelles contraintes

Pour autant, l'Event Sourcing n'est pas une Silver Bullet, qui résoudrait à lui seul l'ensemble des problématiques rencontrées dans le développement logiciel. Aucun patron / architecture / méthode ne l'est, tous amènent des contraintes qu'il faut pouvoir être à même d'embrasser.

La première contrainte est la complexité de l'extraction des données à des fins de recherche. L'Event Sourcing impose de reconstruire un objet depuis l'ensemble des événements qui se sont produits dans la vie de cet objet. Dès lors, rechercher des objets parmi plusieurs milliers d'objets ayant une caractéristique commune, va imposer la reconstruction à la volée de l'état actuel individuel de l'ensemble de ces objets pour pouvoir y appliquer le prédicat voulu. Le contournement le plus répandu est d'associer Event Sourcing et CQRS, afin de tirer parti du Read Side de ce dernier.

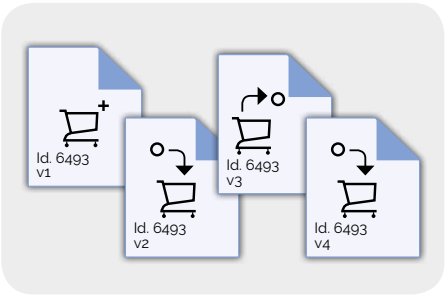
La seconde est la nécessité d'avoir accès aux experts métiers. En effet, comme l'Event Sourcing se base en très grande partie sur les cycles de vie métier des objets, c'est-à-dire l'enchaînement de commandes et d'événements métiers au sein de l'application, la nécessité d'une vision claire sur ces cycles de vie est prépondérante.

Enfin la troisième contrainte, a trait à l'implémentation en elle-même des objets métiers, car il devient indispensable de séparer la notion de décision suite à une commande, c'est-à-dire l'application des règles metiers, de la notion de changement d'état de l'objet, sa mutation. L'implémentation des objets métiers change alors sensiblement par rapport à une approche classique.



Le magasin d'événements, le gardien de la vérité

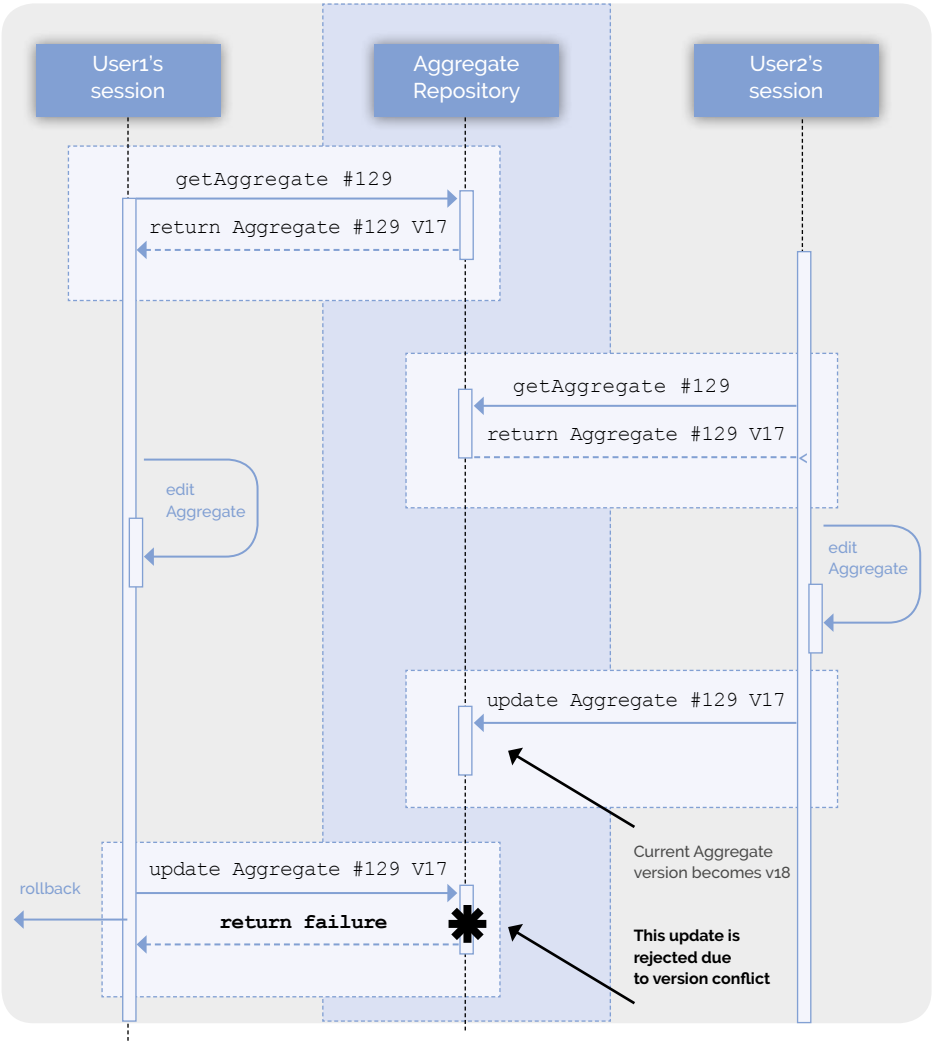
En utilisant Event Sourcing, les données sont persistées dans un Event Store, un magasin d'événements. Son rôle est de stocker une file d'événements métiers pour chaque concept métier dans le système. En considérant les événements comme un ensemble d'informations relatives à l'instance d'un concept, un event store peut être vu comme une base documentaire, où chaque document est lié à l'instance de l'agrégat et est ordonné par date d'occurrence croissante.



L'histoire métier du système ne pouvant être réécrite, le magasin d'événements fonctionne en "ajout seulement" (Append Only). En plus de garantir l'immuabilité de l'historique, le mode de fonctionnement append only permet d'éviter de verrouiller le magasin d'événements lors des lectures et des écritures, entraînant un effet de bord bénéfique sur les performances en écriture. Les performances en lecture sont elles aussi garanties. En effet, l'accès à l'Event Stream d'un agrégat spécifique se faisant par l'identifiant de l'agrégat, récupérer cet event stream consiste à lire tous les documents existants qui référencent cet identifiant.

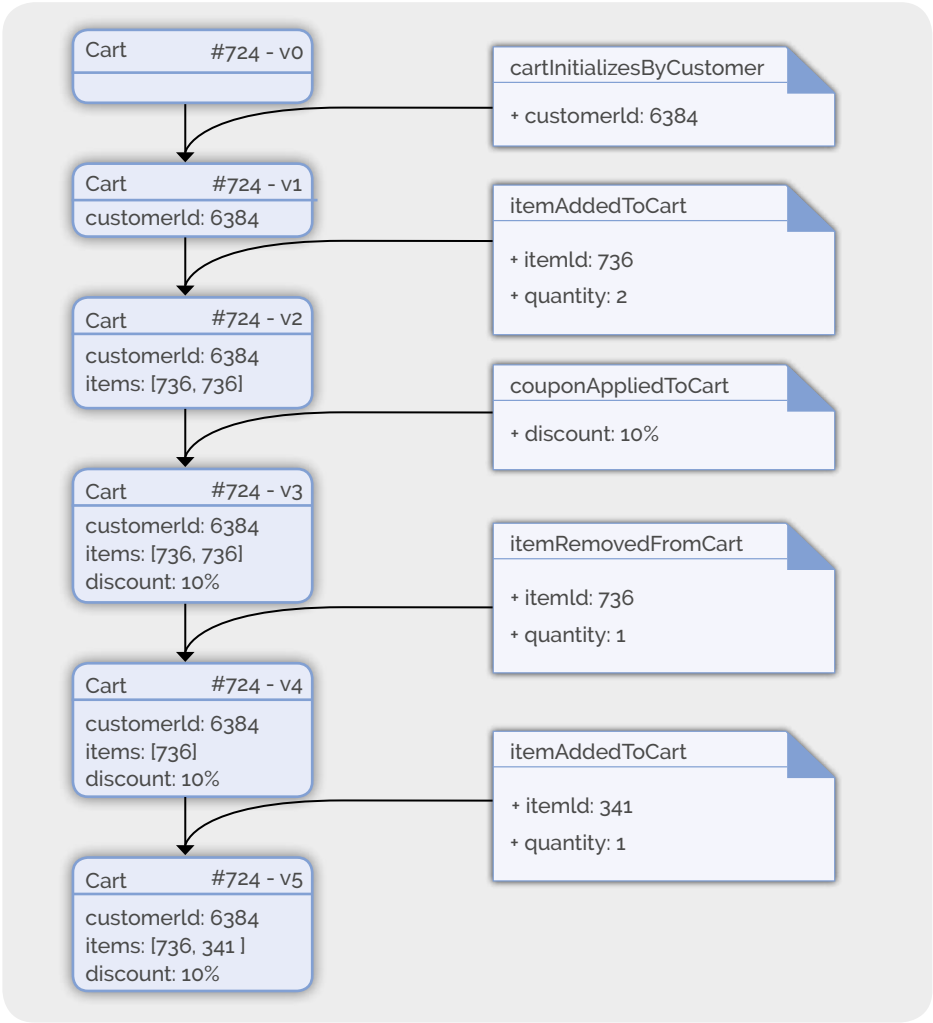
Pour se prémunir de possibles race conditions, un event store implémentera un mécanisme de verrouillage optimiste hors ligne.

Certaines solutions d'events store du marché intègrent un mécanisme de création et de gestion de projections. Les projections au sein d'un système event sourcé seront vues plus en détails dans le chapitre suivant : CQRS & ES (page 21).



Hydratation des agrégats en Event Sourcing

Contrairement aux systèmes utilisant une persistance de l'état de ses objets, où la re-construction d'un agrégat est relativement triviale, dans un système event sourcé, le mécanisme est plus subtil. Le module chargé d'exploiter un agrégat donné commence par créer une instance de l'agrégat dans son état initial, qui n'a jamais eu à prendre en charge une action métier. Puis, ce module récupère l'event stream relatif à l'agrégat fraîchement instancié et applique chronologiquement chacun des événements du stream, via la fonction d'évolution de l'agrégat.



L'application d'événements sur un agrégat ne faisant intervenir aucune logique métier, l'hydratation d'un agrégat est très rapide, y compris lorsque l'event stream est constitué de quelques dizaines d'événements.

Snapshot d'agrégats

Cependant, si un event stream contient tellement d'événements que l'hydratation devient un goulot d'étranglement dans le système, il faut alors envisager l'utilisation de snapshots, c'est-à-dire de captures d'état intermédiaires de nos agrégats.
Il s'agit de stocker périodiquement une version sérialisée d'un agrégat. Cette sérialisation et ce stockage s'effectuent chaque fois qu'un nombre arbitraire d'événements a été appliqué sur une instance donnée d'un agrégat. Idéalement, ce nombre arbitraire d'événements est déterminé par des mesures de performance, et est défini par le seuil où la réhydratation d'une instance d'un agrégat consomme un temps de calcul significatif lors de l'exécution des uses cases de cet agrégat.
Les agrégats étant souvent conçus pour avoir des cycles de vie maîtrisés et bornés dans le temps, l'usage du snapshoting ne sera pas automatique et les concepteurs auront pour rôle, entre autres, de limiter les cas où ce dernier est nécessaire.
La mise en place d'un mécanisme de snapshoting est plus simple qu'il n'y paraît : à chaque fois que le seuil d'événements défini est atteint par l'instance d'un agrégat, ce dernier est sérialisé et stocké sous forme de document portant le numéro d'événement précédent celui qui vient d'avoir lieu. Ce snapshot est stocké en même temps que ledit événement.
Lors de la réhydratation, l'instance de l'agrégat sera créée en déserialisant le snapshot, qui sera ensuite hydraté par les événements portant un numéro supérieur à celui de l'événement porté par le snapshot.

Compensation d'événements

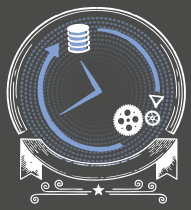
La réalité de ce qui est arrivé à une chose ne pouvant être modifiée, Event Sourcing n'autorise en aucun cas la modification d'un event stream ni des événements composant un event stream. Si un event stream comporte des erreurs, il convient de le corriger en appliquant des compensations. Cela prend la forme d'événements de compensation, ajoutés à la fin de l'event stream. Ces événements sont chargés de corriger les erreurs, mais restent avant tout des événements métiers, dont le nom porte la notion de compensation.

Versionnage des événements

Le métier des entreprises évolue constamment, les processus modélisant ces métiers évoluent eux aussi. Il est par conséquent fréquent de devoir faire évoluer les agrégats et les événements métiers dans un système.
L'historique des agrégats ne devant être modifié sous aucun prétexte, il faut donc mettre en place des stratégies capables de mettre en pratique ces évolutions sans corrompre l'historique.

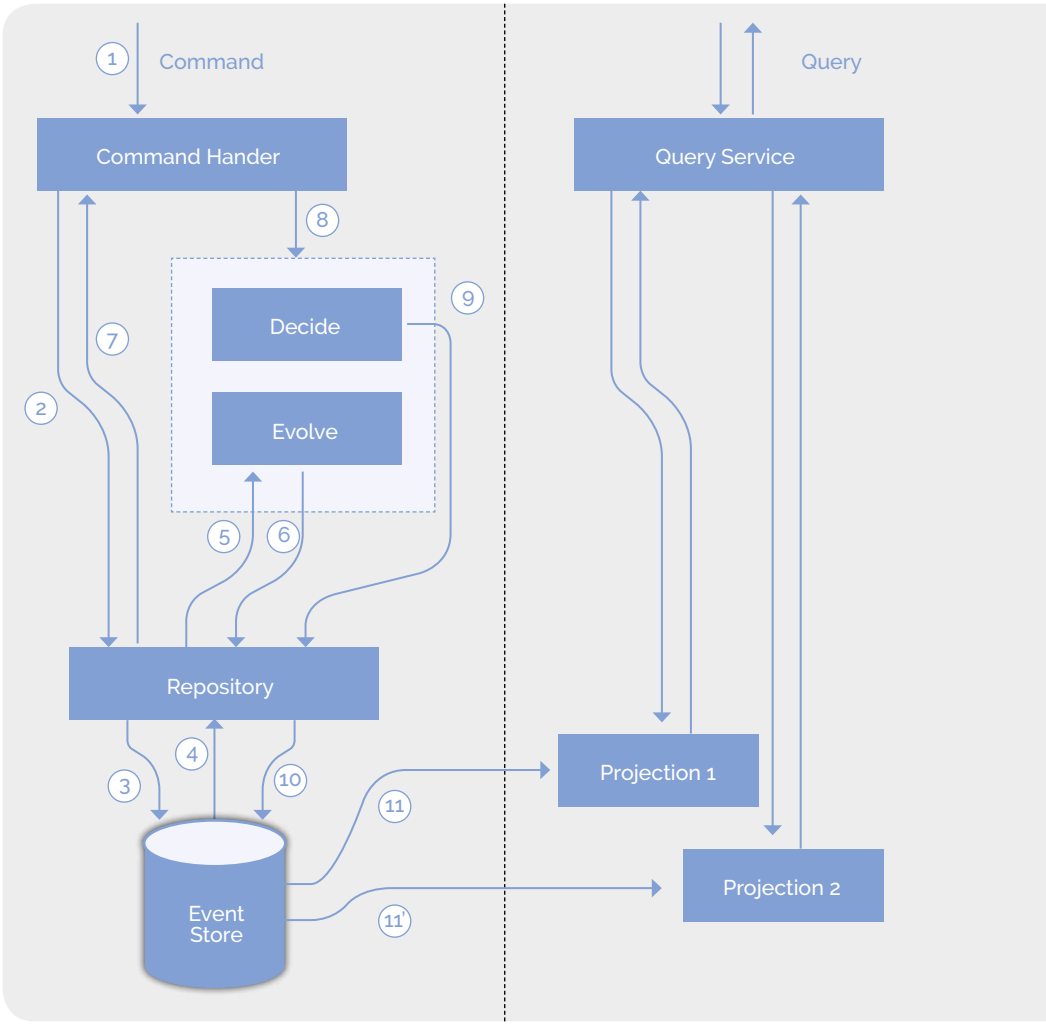
III PARTIE

CQRS & Event Sourcing : le duo gagnant



Deux motifs architecturaux en symbiose

Bien que CQRS et Event Sourcing ne soient pas intrinsèquement liés, et que l'un soit utilisé indépendamment de l'autre, il s'avère que l'association des deux apporte une valeur supérieure à la somme des parties. Chaque motif présente une solution à certaines faiblesses de l'autre.



Le Command Handler reçoit une commande (1), il interroge le repository afin d'obtenir l'agrégat concerné par la commande (2). Le repository récupère tous les évènements concernant cet agrégat (3) et (4) et applique, une par une, toutes les mutations liées à ces évènements (5) et (6). Le repository retourne donc l'agrégat hydraté au commande handler (7) qui peut alors exercer le cas d'utilisation ciblé par la commande afin de prendre une décision métier (8). La logique métier est alors appliquée et un ou plusieurs évènements métiers peuvent être émis et stockés dans l'event store (9) et (10). Les projections concernées par ces évènements sont alors mises à jour (11) et (11') et le query service dispose de données projetées mises à jour.

Des projections facilitées

L'un des principaux défauts de l'Event Sourcing est la complexité d'effectuer des recherches par prédicat sur des ensembles d'agrégats. CQRS répond parfaitement à cette problématique grâce à la formalisation du découplage du Read Side, par le biais des projections. Si un moteur d'Event Store propose ce mécanisme, l'intégration de cette projection dans le Read Side d'une application implémentant CQRS, sera aisée. La construction de ce Read Side sera elle-même triviale, puisqu'elle consistera en une fine couche de services qui accèdent aux données en formatant les projections issues de l'event store.

Limiter l'*impedence mismatch*

Dans le même ordre d'idée, au sein d'un système implémentant CQRS, beaucoup d'événements métiers circulent. Utiliser Event Sourcing en mécanisme de persistance dans le Write Side permet de réduire l'*impedence mismatch*, terme qui désigne la friction entre un modèle métier et sa persistance dans le modèle de données.
Le modèle métier et le modèle de données possèdent des responsabilités très différentes : le premier doit permettre d'adresser les use cases du métier, de la manière la plus efficace possible. Parmi les critères d'efficacité notables, il faut citer l'expressivité du modèle, pierre angulaire de sa maintenabilité et de son évolutivité. Cette expressivité passe par la mise en place

d'une encapsulation forte et d'un découplage des différents concepts selon leur portée transactionnelle.
Le second, le modèle de données poursuit quant à lui d'autres objectifs. Il doit garantir l'intégrité des données et de bonnes performances en lecture et en écriture afin de ne pas devenir un goulot d'étranglement pour l'application.
Ces objectifs différents font que les deux modèles ne sont pas de stricts reflets l'un de l'autre. Par exemple, un objet encapsulant une certaine complexité métier, utilisé par composition au sein d'un autre objet, sera certainement modélisé comme un objet à part entière dans le modèle métier, mais pourra certainement être traité comme un simple champ dans un modèle de données relationnel.
Adopter l'Event Sourcing dans le cadre d'une application architecturée selon les principes de CQRS, c'est avoir l'assurance que le Write Side ne persiste que les valeurs primitives circulant dans le système, c'est-à-dire les événements métiers, plutôt que la dérivée de ces primitives, autrement dit, les états. Cela permet de rendre le modèle métier plus souple et plus apte à recevoir des évolutions, tout en éliminant les fastidieuses étapes de migration de données lors de ces évolutions.

Eventual Consistency

Les deux motifs architecturaux présentés dans ce livre tirent parti de la distinction entre les processus métiers dont la cohérence doit être transactionnelle et ceux qui permettent plus de souplesse en termes de durée.
En reprenant l'exemple d'un site e-commerce, les experts métiers ne souhaitent pas forcément qu'un visiteur soit obligé de s'inscrire pour commencer à ajouter des articles dans son panier. Le fait qu'un visiteur n'ait pas besoin de finaliser son inscription au site pour commencer son shopping est un exemple de frontière transactionnelle souple. Le système s'assurera en revanche que le visiteur dispose d'un compte utilisateur vérifié, avant d'envoyer la commande en préparation d'expédition.

Gestion de l'eventual consistency au niveau UI

1. Désactiver l'UI et la rafraîchir après quelques temps (avec spinner/loader)
2. Interroger à intervalles réguliers le Read Side depuis le controller durant la commande (synchrone) (avec spinner/loader)
3. Simuler que tout s'est bien passé, dans le cas où la bonne exécution est un cas très majoritaire
4. Écran de confirmation (dont la génération poll le query model)
5. Polling côté client

Importance de la bonne conception métier, DDD à la rescousse

CQRS et Event Sourcing ne devraient pas être mis en place sans une justification métier forte. Cependant, cette justification ne viendra pas forcément d'un cahier des charges fonctionnel, mais d'une compréhension solide et profonde du métier qu'adresse l'application. Opter pour l'approche Domain Driven Design permet justement d'adresser la complexité métier de manière très efficace, tant dans les étapes de conception que durant la réalisation.
DDD se concentre en premier lieu sur le langage du métier, les différentes commandes, les événements métiers, les frontières transactionnelles et les cycles de vie des agrégats qui sont identifiés et intégrés tels quels dans le code. L'approche CQRS + Event Sourcing reprend alors de manière très claire ces éléments, et le système dans son ensemble devient naturellement une extension du métier.

Notre avis : ***ARCHITECTURE LOGICIEL INTERNE DANS LE MODÈLE MÉTIER***
Bien que le Write Side puisse être architecturé de la manière qui vous convient le mieux, SOAT vous recommande fortement d'adopter le pattern "Ports & Adapters", qui permet d'isoler au mieux votre code métier, en l'isolant des considérations d'infrastructures.

Découvrir et identifier les processus métiers à l'aide d'Event Storming

Comprendre le métier est la condition *sine qua non* pour réaliser des applications utiles et efficaces. Il n'est cependant pas simple d'obtenir cette compréhension. Parmi les outils facilitant le partage autour du métier à adresser, l'Event Storming, mis au point par Alberto Brandolini, fait partie des plus efficaces.
Il permet dans un temps optimal, de récupérer un maximum de connaissances sur une partie ou sur l'ensemble d'un métier en question. L'organisation de cette activité nécessite beaucoup d'efforts car la présence d'experts métiers et de développeurs est requise. Pour plus d'informations sur cette activité, vous pouvez vous référer au livre Event Storming écrit par Alberto brandolini.

Conclusion

L'utilisation combinée des motifs CQRS et Event Sourcing s'avère particulièrement efficace pour résoudre les problématiques récurrentes que l'on rencontre au sein des applications métiers complexes. Que ce soit en termes de prise en compte du métier, de maintenabilité ou de mise à l'échelle, les bénéfices sont multiples.

Cependant, il faut bien garder à l'esprit qu'il ne s'agit pas d'une recette miracle qui soignerait tous les maux. Le niveau technique global de l'application ayant tendance à augmenter, les équipes intervenant sur des projets CQRS Event Sourcés doivent être formées et faire preuve d'une maturité plus grande que sur des applications architecturées en couche classique. La courbe d'apprentissage tend également à être plus lente pour des développeurs moins expérimentés.

Reste qu'une bonne organisation du code peut faire ressortir aisément les tenants et aboutissants métiers de l'application, à condition de collaborer très étroitement avec les experts métiers.

Ces patterns mettent l'accent sur le code métier, et permettent d'en améliorer la maintenabilité. Dans ce cadre, nous vous recommandons fortement d'utiliser une approche centrée sur le métier, comme DDD.

Cette approche mettant l'accent sur la transactionnalité des processus métiers, DDD permet de comprendre intrinsèquement les topologies de cohérences nécessaires, afin de concevoir le plus efficacement possible les agrégats mis en œuvre dans le système et de tirer le meilleur parti des opportunités de cohérence finale.

Vous l'aurez compris, nous vous recommandons d'explorer ces patterns, au vu des bénéfices attendus dans vos applications traitant une complexité métier élevée.

En portant la séparation claire de vos enjeux métiers entre plusieurs éléments dédiés et en axant la persistance sur vos données primitives, CQRS et Event Sourcing vous permettront de retirer plus de valeurs de vos systèmes.

Auteurs

CQRS & Event Sourcing : Edition 2018

Antoine SAUVINET

Architecte, Coach Technique
et Formateur chez SOAT

Fabien WIRIG

Coach Technique
et Formateur chez SOAT

Sepehr NAMDAR FARD

Craftsman, Coach technique
et Formateur chez SOAT

CONCEPTION GRAPHIQUE ET RÉALISATION : MAP ADVERTISING

SOAT

www.soat.fr - blog.soat.fr

Sequana 1
89 quai Panhard et Levassor
75013 Paris
01 44 75 42 55