L'architecture hexagonale (ou *ports* & *adapters*) a été formalisée en 2005 par Alistair Cockburn.

Depuis quelque temps, j'utilise avec plaisir une variante de cette architecture dans des applications Spring Boot. J'ai aussi participé à différents lives twitch ou nous avons codé des petites applications tirant parti de cette architecture : Borestop et PadBowl.

Cet article est une présentation de la manière dont j'utilise cette architecture et de ce pourquoi je l'apprécie. J'utiliserai donc souvent la première personne mais je n'en suis arrivé à ces conclusions que grâce au travail des personnes avec lesquelles nous avons expérimenté tout ça !

Où et quand l'utiliser ?

Elle peut être utilisée avec des modèles anémiques mais je ne l'utilise pas dans ce contexte, donc je ne pourrais pas en parler...

Comme beaucoup, j'ai découvert cette architecture en m'intéressant au DDD. De fait, j'ai tendance à l'utiliser pour protéger des *domain models* riches. Dans ce type d'environnements, on va trouver une architecture hexagonale par *Bounded Context*.

> Un *Bounded Context* (contexte borné) permet le traitement autonome d'un processus métier. On sait qu'on passe d'un *Bounded Context* à un autre quand un même terme ne veut plus dire la même chose, qu'on change d'équipe, de base de code, ...

Pour moi, elle prend donc tout son sens avec une équipe pouvant *designer* des *domain models* riches. Même dans cet environnement, tous les modules (*packages*) n'ont pas besoin de ce type d'architecture : elle a un coût, il faut donc l'utiliser uniquement lorsque c'est rentable.

L'architecture

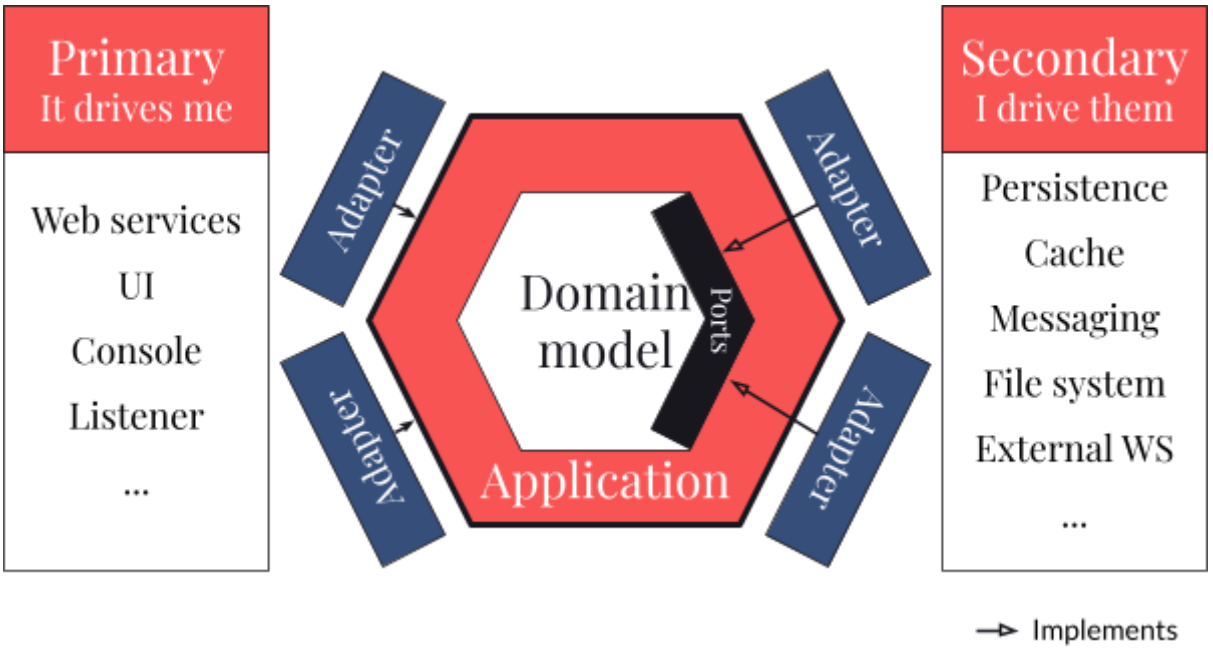
La première fois que j'ai croisé cette architecture, c'était dans Implementing Domain-Driven Design de Vaughn Vernon. Mais bon, comme je n'ai pas apprécié ce livre, je suis passé à côté. Même si la manière dont elle est utilisée dans ce livre ne me plait pas (du tout), je suis tout à fait d'accord avec la remarque de l'auteur juste avant de la présenter :

> *"Architecture is important, but architectural influences come and go. Remember to prioritize correctly, placing more emphasis on the domain model, which has greater business value and will be more enduring."*

-> L'architecture à son importance mais il faut prêter une plus grande attention au *domain model* qu'à l'architecture.

En fait, je pense que c'est pour cette raison que j'aime beaucoup la variante que j'utilise : elle me permet de traiter les problèmes responsabilité par responsabilité, elle ne vient pas "m'agresser" au quotidien !

Quand je dois schématiser cette variante je fais à peu près ça :



Bon, vous n'êtes probablement pas vraiment avancés... allons voir bloc par bloc ce qui se passe dans cet hexagone !

Le Domain Model

Commençons par le centre puisque cette architecture est bel et bien centrée sur le *domain model* :

Il ne dépend de rien et tout dépend de lui. Ce point est essentiel : aucune dépendance à aucun framework dans le *domain model* ! Pas de dépendance, non plus, à d'autres éléments de l'hexagone. Il doit permettre une réponse saine aux besoins Métier, il ne s'occupe pas du câblage technologique.

Ce détachement technologique rend le code du *domain model* très simplement testable, on pourra donc efficacement (et sans connaissances technologiques poussées) obtenir des *designs* élégants et pragmatiques en TDD (//TODO: lien : Ça c'est TDD).

Comme il est au centre, on commencera très souvent un ticket en lui apportant les modifications nécessaires, le reste suivra naturellement !

Tout dépend de lui, donc, s'il contrôle vraiment les données, ne manipule pas de primitives et n'a jamais de `*null*` ben... ce sera le cas dans tout le *Bounded Context* (et c'est une super bonne nouvelle !).

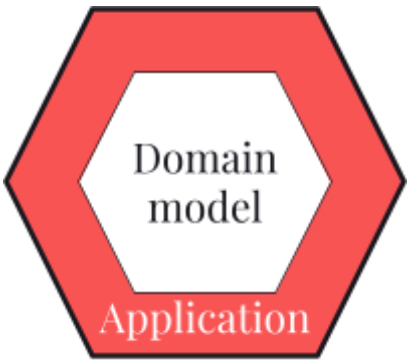
On va donc commencer par faire des objets, pas des data classes, on n'hésitera pas non plus à utiliser des patterns de construction pour être systématiquement dans des états cohérents. Si un objet peut être `*null*` on utilisera un `*Optional*`, s'il s'agit d'une `*Collection*` on aura une `*Collection*` vide. De cette manière, notre *domain* exprimera clairement ce qu'il fait et "forcera" les traitements nécessaires à la prise en compte de tous les cas !

> Même s'il ne manipule pas directement de primitives (int, boolean, String, ...) il est souvent pratique de construire les *domain models* depuis des primitives. Deux stratégies sont possibles : construire depuis des primitives et en faire des types dans le *domain* ou construire le *domain* directement depuis des types. Pour avoir essayé les deux approches, je préfère maintenant construire les objets du *domain* depuis des primitives (commit) même si, parfois, pour une même donnée, les deux stratégies seront exposées.

Les premières fois que vous allez coder un *domain model* vous allez probablement exposer tous les attributs et ne pas faire grand-chose de plus. À mon sens, c'est pour cette raison qu'il est essentiel de faire ce *domain model* en TDD : de cette manière vous pourrez faire émerger un *design* pragmatique répondant vraiment aux besoins.

Le layer application

Dans la variante présentée ici, il y a un *layer application* autour du *domain model* :



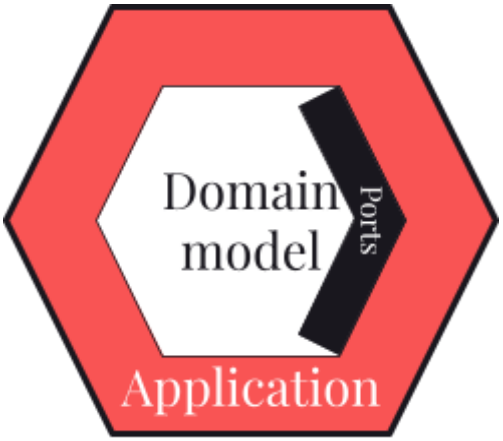
Ce *layer* ne fait aucun traitement métier (puisque c'est le *domain model* qui les fait) ! C'est souvent un des points les plus complexes quand on passe d'une architecture avec un *@Controller* qui appelle un *@Service* qui, lui-même, appelle un *@Repository*. **Les méthodes ici font rarement plus de 4 lignes** (avec un seul point par ligne).

> Il est possible d'avoir des services dans le *domain model*. C'est pour cette raison que les services dans le *layer application* sont nommés **ApplicationService* : pour éviter toute confusion.

Les *ApplicationServices* dans ce *layer* ne doivent pas être confondus avec les Services auxquels nous sommes parfois habitués. Les *ApplicationServices* sont responsables :

- De l'orchestration simple (pas de Métier) des opérations ;
- De la gestion des transactions ;
- De la gestion des authorizations (que je fais avec kipe).

Pour faire son orchestration, le *layer application* doit pouvoir communiquer avec "l'extérieur", c'est pourquoi on ajoute des *ports* dans le *domain model* (nous verrons un peu plus loin comment ces *ports* permettent la communication) :



> Nous avons longtemps tâtonné sur l'endroit où devaient se trouver les *ports*, mais il n'y a pas de doute possible : le *domain model* a parfois besoin des *ports*, il les portent donc forcément.

Ces *ports* sont des interfaces exposant des opérations métier (et pas des opérations techniques). Ils suivent le principe de ségrégation des interfaces.

Bien, prenons un exemple concret d'un *ApplicationService* :

```
```java
@Service
@Transactional
public class InvoicesApplicationService {

  private final InvoicesRepository invoices;

  public InvoicesApplicationService(InvoicesRepository invoices) {
    this.invoices = invoices;
  }


  @PreAuthorize("can('claim', #invoice)")
  public void claim(InvoiceId id) {
    Invoice invoice =
invoices.get(id).orElseThrow(UnknownInvoiceException::new);
```
```

```
    invoice.claim();

    invoices.save(invoice);
  }
}
```

Avec :

- `Invoice` et `InvoiceId` : des objets du *domain* permettant la manipulation de factures ;
- `InvoicesRepository` : un *port* permettant la manipulation de la persistance des `Invoice`.

Alors, certes `InvoicesApplicationService` est annoté *@Service*, mais c'est un service très différent de ceux que l'on croise habituellement ! Les traitements métier sont délégués au *domain model* et le service ne porte que les 3 responsabilités évoquées plus haut (*orchestration*, *transactions* et *authorizations*).

Dans certains cas (commit), les opérations Métier vont créer des événements métier (on parle de *Domain Events* en DDD) et il faudra envoyer (*dispatch*) ces *events*. Après avoir expérimenté différentes stratégies je fais maintenant :

- Des méthodes dans le *domain model* qui renvoient les *events* ;
- Des *ports* dédiés au *dispatch* d'events ;
- Un appel au dispatch dans les *ApplicationService*.

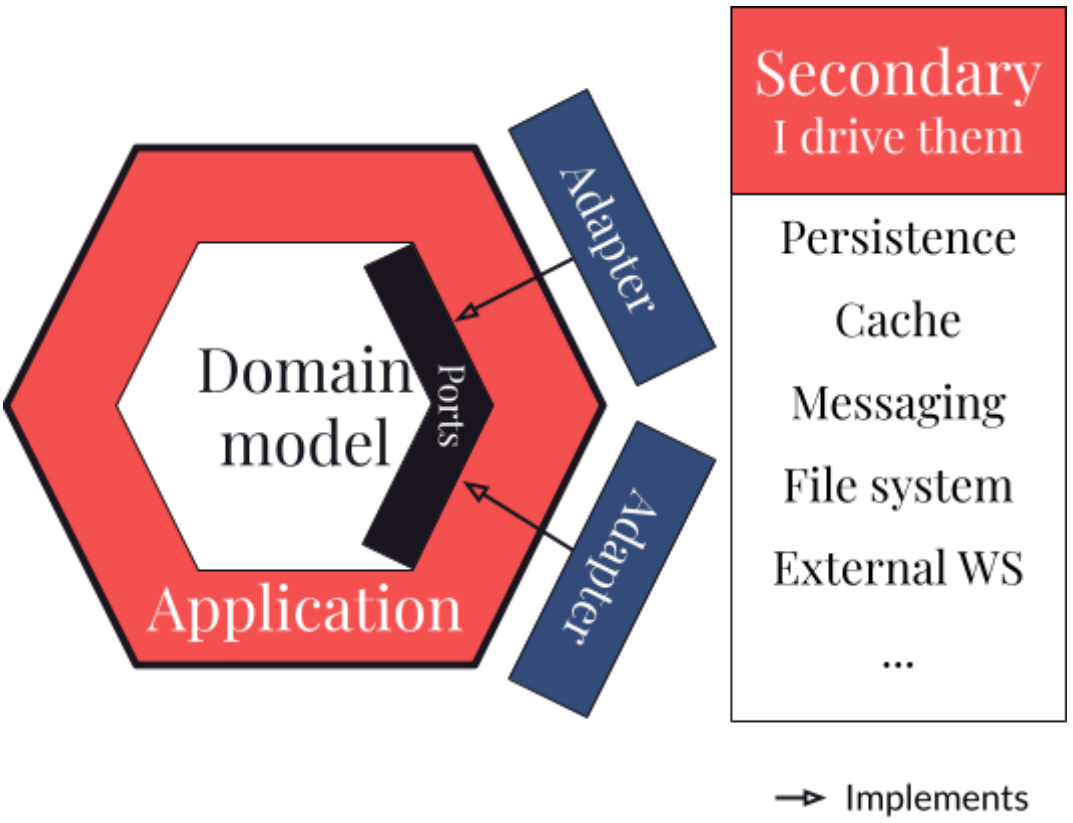
> Il existe bien d'autres stratégies, on peut injecter le *port* de *dispatch* dans le *domain model* (en paramètre de la méthode créant les événements) ou utiliser des *dispatchers static*. C'est à vous de choisir la manière de faire qui vous convient le mieux !

Quand vous allez commencer à utiliser cette architecture, il est fort probable que les ApplicationServices continuent de faire des traitements métier, même simples. Il est très important de faire la chasse à ce métier qui n'a rien à faire là pour bénéficier des avantages de ces approches !

Les secondary adapters

Nous venons de parler des *ports*, en réalité nous venons de parler des *secondary ports* ! Il est donc logique de parler des *secondary adapters* qui vont **implémenter nos ports**.

Les *secondary adapters* sont aussi appelés *output adapters*. Ce sont les implémentations pour les éléments que notre *Bounded Context* va piloter : persistance, cache, file de messages, système de fichiers, etc... :



Les *secondary adapters* n'ont qu'une chose à faire : avoir des interactions saines avec les éléments externes que notre *context* manipule. Le découpage fort entre les objets Métier et la manière dont ils vont être persistés, mis en cache, ... permet des modélisations vraiment adaptées aux deux mondes !

> Dans la variante présentée ici, les *transactions* ne sont pas gérées dans les *secondary adapters* puisqu'elles sont prises en compte dans les *ApplicationServices*. Cette adaptation facilite les traitements du type : récupération d'un objet persisté -> opération métier -> sauvegarde.

Les *secondary adapters*, pour un *context* donné, ont une très forte cohésion. Dans la très (très) grande majorité des cas, tous les objets dans les modules (*packages*) *secondary* seront en visibilité *package*. De cette manière on limite le bruit dans le reste de l'application.

Après avoir essayé différentes stratégies, je fais maintenant les conversions depuis et vers les objets du *domain model* dans les objets utilisés par mes *secondary adapters* :

```java
@Entity
@Table(name = "invoice")
class InvoiceEntity {
  // Fields and constructor

  static InvoiceEntity from(Invoice invoice) {
    // Build an InvoiceEntity from an Invoice
  }
```

```
   Invoice toDomain() {
    // Build an Invoice from an InvoiceEntity
  }

  // Getters, setters, hashCode and equals so hibernate is happy

}
```

Ce qui donne des implémentations très simples des *secondary adapters* :

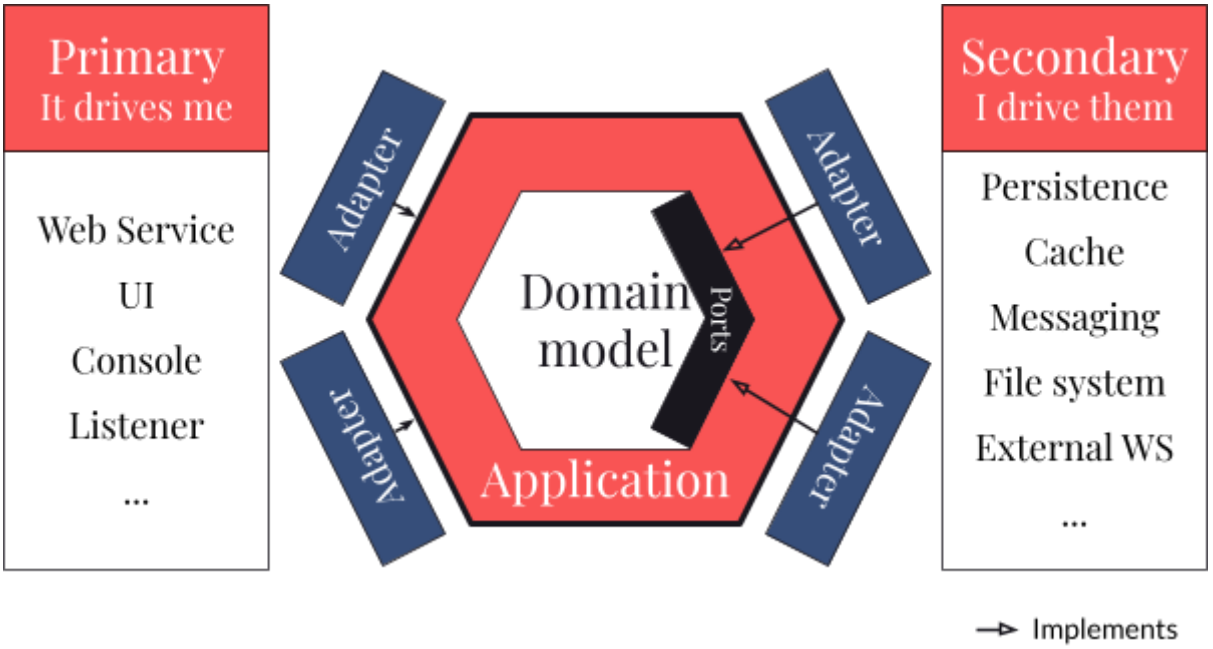
```java
@Repository
class PostgreSQLInvoicesRepository implements InvoicesRepository {
  private final SpringInvoicesRepository invoices;

  public PostGreSQLInvoicesRepository(SpringInvoicesRepository invoices) {
    this.invoices = invoices;
  }

  @Override
  public void save(Invoice invoice) {
    Assert.notNull("invoice", invoice);

    invoices.save(InvoiceEntity.from(invoice));
  }

  @Override
  public Optional<Invoice> get(InvoiceId id) {
    Assert.notNull("id", id);

    return invoices.findById(id.get()).map(InvoiceEntity::toDomain);
  }
}
```

> Dans les premières semaines d'un projet, il est tout à fait possible (voire conseillé) de faire des implémentations de *secondary adapters* faisant des opérations en mémoire ! Ces implémentations peu coûteuses serviront à découvrir les réels besoins. Une fois ces besoins vraiment connus, on pourra choisir les outils les plus adaptés pour y répondre.

Quand on commence à travailler avec ce type d'architecture, on a tendance à commencer par les *secondary adapters* pour revenir en "terrain connu". Il faut vraiment prendre l'habitude de commencer par le *domain model* en ne pensant pas tout de suite à la manière de le persister. C'est cette approche qui remet le métier au centre des discussions et des préoccupations, les *adapters* ne sont qu'un détail d'implémentation !

Les primary adapters

Les *primary adapters* sont les implémentations pour prendre en compte les événements externes qui vont piloter notre *context* : Web Services, Listeners de message, ...



Dans cette variante, ils interagissent directement avec le *layer application*, on injecte donc directement les *ApplicationServices* dans nos *primary adapters*.

Les avantages dont ils bénéficient sont les mêmes que les *secondary adapters* (forte cohésion, responsabilité claire et limitée, ...). On va donc retrouver des patterns très similaires ici, même si les technologies utilisées vont être très différentes.

> Je fais régulièrement des *primary adapters* qui peuvent sembler étranges : des implémentations en Java. Ce sont ces adapters qui permettent le questionnement depuis d'autres *Bounded Context* (qui auront alors des *secondary adapters* appelant ces implémentations). De cette manière, je peux pré-découper un monolithe pour préparer un éventuel découpage en microservices.

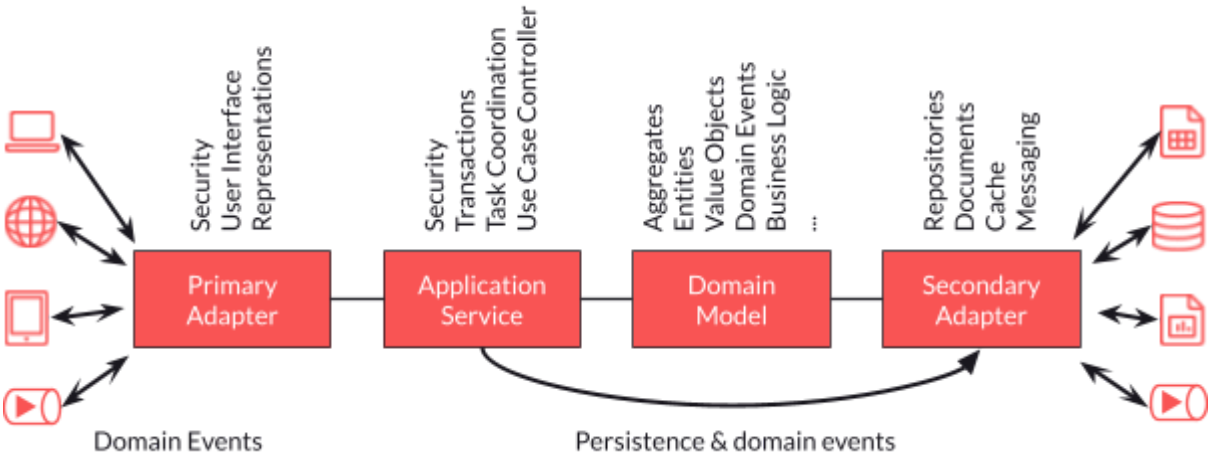
La gestion des exceptions

Avec ce type d'architecture, la gestion des exceptions peut paraître complexe ! Elle est en fait très simple : comme pour tous les objets, on crée les exceptions au bon endroit (les exceptions du *domain* dans le *domain*, etc). D'ailleurs, dans une architecture hexagonale les exceptions seront souvent en visibilité *package*.

En gardant des *unchecked exceptions*, elles remonteront naturellement jusqu'au *layer* devant les prendre en compte. Pour une erreur qui doit être affichée à l'utilisateur on pourra avoir un @ControllerAdvice dans un *primary* (il pourra ressembler à ce qui est décrit dans cet article).

Pour conclure

La première fois que j'ai croisé une architecture hexagonale, c'était un livre de Vaughn Vernon que je n'ai pas aimé. Quand j'ai compris ses forces et comment l'utiliser, c'était dans un livre de Vaughn Vernon que j'ai beaucoup aimé : Domain-Driven Design Distilled. Le schéma qui m'a permis d'y voir plus clair est celui-là :



Même si, avec le temps, j'ai apporté quelques modifications à cette séquence, elle reste très proche de celle décrite dans cet article et que j'utilise au quotidien.

Pour représenter cette architecture, les modules (*packages*) des applications sur lesquelles j'interviens sont organisés de cette manière :

```
invoice
- application
- domain
- infrastructure
  - primary
  - secondary
```

Dans cet exemple "invoice" est le context, ce nom change pour chaque context.

> Vous pouvez trouver un exemple de *Bounded Context* organisé de cette manière sur PadBowl dans le *Bounded Context game*.

Avec le temps et la pratique, je ne rencontre pas de blocage ou de difficultés majeures avec cette architecture. Sur plusieurs produits où elle a été mise en place, il y a eu des changements d'équipe et cette architecture ne se contente pas de rester : elle est appréciée au quotidien !

Ce n'est cependant pas une *silver bullet*, elle n'est pas adaptée à tous les contextes (ni à toutes les applications). Je ne peux que vous conseiller d'essayer si vous pensez être dans un contexte s'y prêtant.

Aujourd'hui, avec le temps, l'énergie et la qualité gagnées au quotidien sur des produits avec des traitements métier, je ne me vois pas passer à une autre architecture !