

## Domain Driven Design

- **DDD:** C'est une façon de regarder le logiciel de haut en bas: on monte de bas en haut en ajoutant à chaque fois un niveau d'abstraction
- **Stratégie design:** Réflexion entourée et centrée

Tels

C'est le cadre dans lequel apparaît un mot ou une déclaration qui détermine sa signification

- Ubiquitous language: langage structuré autour du "domain model" pour connecter les domaines entre eux.
- Bounded context: description des limites d'un contexte
- Context Map: Entity Relationship Types core:
  - \* Published language: The interacting BCs agree on a common language by which they can interact with each other
  - \* Open host service: BC specifies a protocol by which any other BC can use its services (ex: Useful web service).
  - \* Shared Kernel: 2 BCs use a common kernel of code (ex: library) as a common lingue-franca, but otherwise do their own stuff in their own way.
  - \* Customer / Supplier: One BC uses a service of another BC, but is not a stakeholder (Customer) of that other BC (it can influence the services of that BC)
  - \* Conformist: (A) same as customer / supplier, but conforms to the protocols / APIs provided by that BC
  - \* Anti-corruption Layer: (A) Same as customer / supplier but aims to minimize impact from changes in the BC it depends on by introducing a set of adapters.

- **Tactical Design:** les détails de l'implémentation ; composants dans un BC; change durant le développement d'un produit
  - Ressources techniques utilisées dans la construction du domaine model, appliquées pour travailler dans un seul BC

AOP:

- Architecture en couches: chaque couche possède des responsabilités spécifiques
  - Il existe parfois des traitements qui concernent plusieurs couches
  - ⇒ Préoccupations Transversales: Logging - security - monitoring - communications - Exception handling - Performance
- (-): dépendance des objets transversaux ; Pluriers appels directs module technique depuis des modules indiens ; Besoin de /1

modifie toutes les classes en absence d'une structure plus abstraite  
⇒ Solution: AOP: paradigme de programmation qui propose de séparer cette technique du code métier de l'application

- Description:
  - \* Aspect: unité de développement
  - \* Un aspect assure les fonctionnalités transversales dans la注入器 dans le code fonctionnel
  - \* Un aspect est greffé à une classe / procédure pour lui fournir ses fonctionnalités
  - \* **⚠️** C'est un complément pour les autres paradigmes; ne les remplace pas

- Aspect: éléments majeurs:

- \* Joinpoint: où il est possible d'invoquer le griffon

- \* Juncut: où s'exécute exactement le traitement

- \* Advice: le griffon: le traitement qui sera intégré

- \* Aspect: contient les traitements techniques qui seront intégrés à des juncuts

- \* Weaving: Tissage: insertion des aspects

(+): Réutilisabilité - Facilité de maintenance - Amélioration de la qualité de code, spécialisation et indépendance lors du développement

(-): Difficulté d'analyse du code lors des phases de mise au point des logiciels, débogage - test ... - Absence de normalisation: plusieurs approches et implémentations

## Web Services: REST vs SOAP

- Service Web: protocole d'interface informatique permettant la communication et l'échange de données entre applications et systèmes hétérogènes dans des environnements distribués

- WS REST:

- \* Exposent ses fonctionnalités comme un ensemble de ressources identifiables par un URI accessibles par le protocole HTTP

- \* REST :
    - met l'accent sur l'évolutivité des interactions et composants, généralité des interfaces, le déploiement indépendant des composants et des composants intermédiaires

- Architecture

(Representation Söök Transfer)

- Réduire la latence des interactions

- Renforcer la sécurité

- Encapsuler les systèmes hérités.

- \* Quand?: - Bande passante limite

- les informations sur les objets n'ont pas besoin d'être communiquées au client

- \* Quand ne pas?:

- contrat strict entre client et serveur

- Exécution et transaction impliquent plusieurs appels.

## WS SOAP:

- \* Exposent leur fonctionnalité sous la forme de services exécutables à distance.
- \* Reposent sur les standards SOAP et WSDL pour transformer les problématiques d'intégration héritées du monde middleware en objectif d'interopérabilité
- \* **SOAP:**
  - rend les données disponibles sous forme de services "verb+nom"
  - (ex: getUser)
    - Utilisé dans le monde des entreprises où la communication entre ≠ services doit se conformer à un ensemble de règles et d'entités
    - Plus strict que REST

(+):

- + Fonctionne au-dessus de tout protocole de communication (en application)

(-):

- + Beaucoup de bande passante pour communiquer des métadonnées

\* Sécurité + autorisation font partie du protocole.

\* XML uniquement

\* Peut être entièrement décr. par WSDL

\* Difficile à mettre en œuvre et impraticable (web/mobile)

\* Approche formelle

\* Les infos. sur les objets sont communiquées au client

\* **Quand?**:

- contact formel
- des clients doivent avoir accès aux objets dispo. sur les serveurs.
- Maintenir un état

\* **Quand ne pas?**:

- Bande passante limitée  
- Vous souhaitez que la majorité des développeurs utilisent facilement votre API

## Restful WebServices + JSON

• Restful WebServices nécessitent :

- Architecture client - serveur.
- Requêtes statiques  $\Rightarrow$  minimisation des ressources système
- Héritage de cache élimine certaines interactions client-serveur partiellement ou même totalement améliorant l'extensibilité et la performance du système (Proxy servers - for HTTP).
- Interface uniforme: identification des ressources dans les requêtes - Granularité des ressources par des représentations - Tiersages auto-déscriptifs
- Architecture multi couches:
- code à la demande

## Règles de conception des API REST:

- \* Accès aux ressources: URI simple et intuitive. /users/2
- \* Méthodes HTTP comme identifiant des opérations (Get, Post, Put, Delete)
- \* Réponses HTTP comme représentation des ressources. (XML, JSON - HTML)
  - + Gestion des erreurs HTTP
- \* Les liens comme relation entre ressources.
- \* Un paramètre comme jeton d'authentification (ex: JWT)

## JSON: JavaScript Object Notation:

- Format d'échanges de données ; indépendant des langages de programmation, mais utilise des conventions communes à tous les langages.

Ex: [ {

"num": "Sasha",

"id": 5

}, ... ].

## REST :

- (+):
  - Intégration des systèmes et des langages

- Architecture scalable: répartir les requêtes sur plusieurs serveurs statiques

- Facile à comprendre et implémenter

- (-):
  - Sécurité restreinte

- Client doit conserver des données localement (stateless)

## Microservices :

↳ une architecture et une approche de développement logiciel qui consiste à décomposer les applications en éléments plus simples et indépendants les uns des autres

⇒ Approche plus adaptée pour les équipes suivant les méthodes agiles : facilite le dev, test, déploiement et maintenance des systèmes d'applications

Décomposer l'application en petits services, appelés microservices qui sont parfaitement autonomes et qui exposent une API REST que les autres microservices pourront consommer. ≠ Timbrific

- SOA: Service-Oriented Architecture : à une partie d'entreprise, tandis que l'architecture des microservices à une partie d'application

## Caractéristiques de l'architecture microservice:

- Développement / déploiement indépendant
- Langage neutre
- Mise à l'échelle précise

- (+):
- Décentralisation
  - Automatique
  - Continuous delivery
  - Agilité
  - Decentralized governance
  - Componentization
  - Business Capabilities
  - Responsibility

- (-):
- Augmentation du trafic réseau
  - Augmentation de la complexité (plus difficile à développer)
  - Coût de latence augmente
  - Coût serialization/deserialization
  - Test / déploiement plus complexes

### Command Query Responsibility Segregation Pattern:

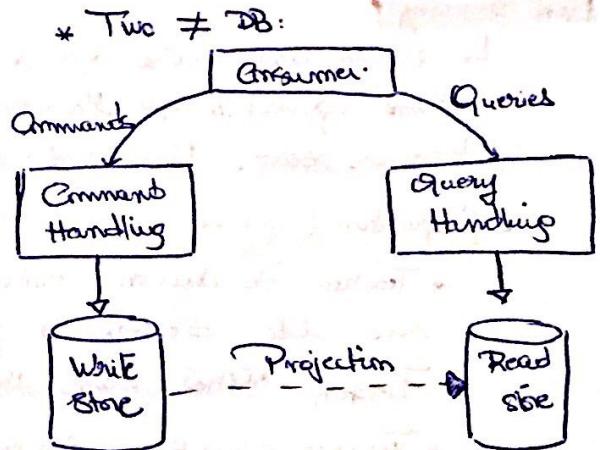
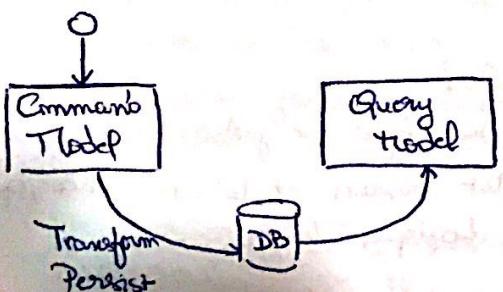
As apps become more complex, handling detailed queries and validation rules also grow in complexity → The conventional CRUD data model could become cumbersome to implement and maintain.

#### • Solution? CQRS Pattern.

- Separates read and update operations for a data store preventing merge conflicts  
→ Enhancing performance, scalability and security
- \* Commands: methods that only perform an action
- \* Queries: methods that read and return data without modifying it.

#### • Database Usage:

- \* One DB: for apps that require synchronous processing and immediate results.



#### • CQRS Implementation Aspects:

- \* Task-Based UI:
- \* Command Processing: When creating a new resource, the ID of the newly created resource is not returned because the read and write models are separated.

Command → Command Bus → Command Handler.

#### \* Synchronous / Asynchronous processing:

- Synchronous: an immediate response is returned to the user.
- Asynchronous: data is written to one source and then is asynchronously processed, then transferred to the read DB

\* Domain Events: listen for certain events during the execution of a transaction to trigger other actions  $\Rightarrow$  decouple the consequence of an action with the action itself.

### \* When? :

- Collaborative domains where the same data is accessed in parallel
- Number of reads > Number of writes
- Team for the write model and another team for the read model/UI
- Business rules change regularly
- Integration with other systems (Event Sourcing)

- (+):
- Facilitates dev workload distribution
  - Maintenance
  - Optimized DB design
  - Scalability
  - Enhanced security

### \* When not? :

- Business rules are simple
- A simple CRUD-style user interface and data access operations are sufficient.

- (-):
- Added complexity
  - Eventual consistency

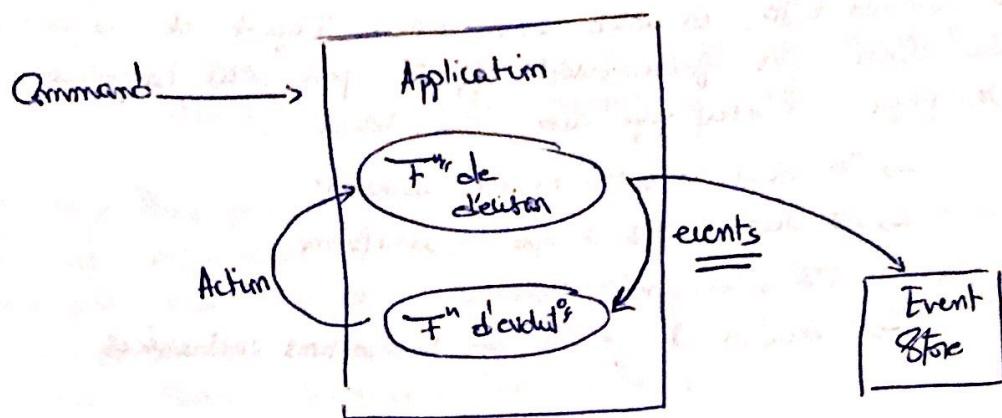
## Event Sourcing :

$\hookrightarrow$  pattern d'architecture qui se concentre sur la séquence de changements d'états d'une application qui l'a amenée dans l'état courant (ex: solde du compte bancaire : état ; transaction : séquence)

### • L'application (système) est composée de 4 fonctions :

- \* Fonction de décision: contient l'ensemble des règles de gestion : elle produit une liste d'événements à partir de l'état courant et la commande, que
- \* Fonction statique dans l'évent élément  $\Rightarrow$  Logique Tertiaire
- \* Fonction d'évolution: fait muter l'état courant : ne contient aucune logique métier.
- Commande: intervention de l'extérieur (ex: débité compte); autosuffisante o.o.d., elle contient tous les paramètres nécessaires à sa description
- Évenement: un fait qui s'est produit dans le passé; autosuffisant; immuable (ne peut pas changer)
- Event store: il est le plus souvent persistant afin de permettre les reprises des données
- Action: une réaction initiée par l'application; autosuffisante; déclenche un second tour de boucle, identique au premier

Effet de bord: chaque application se retrouve dans un état stable, l'ensemble des événements menant à cet état sont transmis à l'extérieur du système, informant ce qui s'est passé à l'intérieur.



- (+): - audit efficace et rapide
  - Facile contribution à l'analyse
  - Facilité à reprendre les données
  - Synchroniser/notifier plusieurs applications en même temps, lorsqu'associé avec CORS.

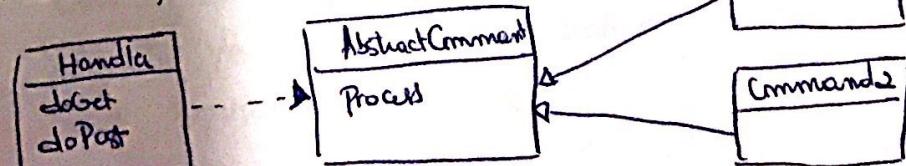
- (-): - Beaucoup d'espace disque
  - Durée de démarrage longue
  - Event sourcing + CQRS: structure complexe.

## EJB Design Patterns:

- Design Patterns EJB généraux:
  - \* Business interface: Toute classe bean possédant une méthode doit implementer une interface qu'on définit.
  - \* Coarse-grained remote interface: Plusieurs appels distants sur le réseau affectent la performance  $\Rightarrow$  faire les appels distants en mode "batch"
  - $\Downarrow$  Loss of security and transaction support related to particular values in this approach

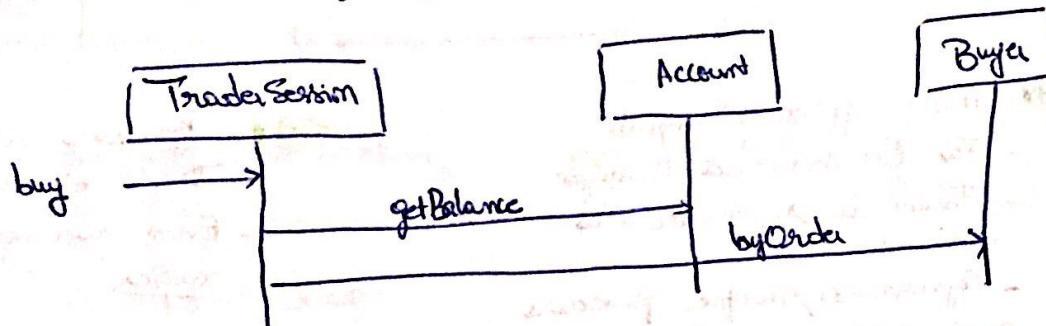
- Design Patterns EJB côté Client:
  - \* Service locator: Un client EJB doit localiser un objet bean de l'entreprise pour l'utiliser après pour trouver un objet ou créer ou supprimer d'autres beans.

- Design Pattern de présentation:
  - \* Front controller: Problèmes de sécurité, de duplication de code, de couplage fort... lorsque l'on utilise plusieurs contrôleurs.



## • Design Patterns d'application:

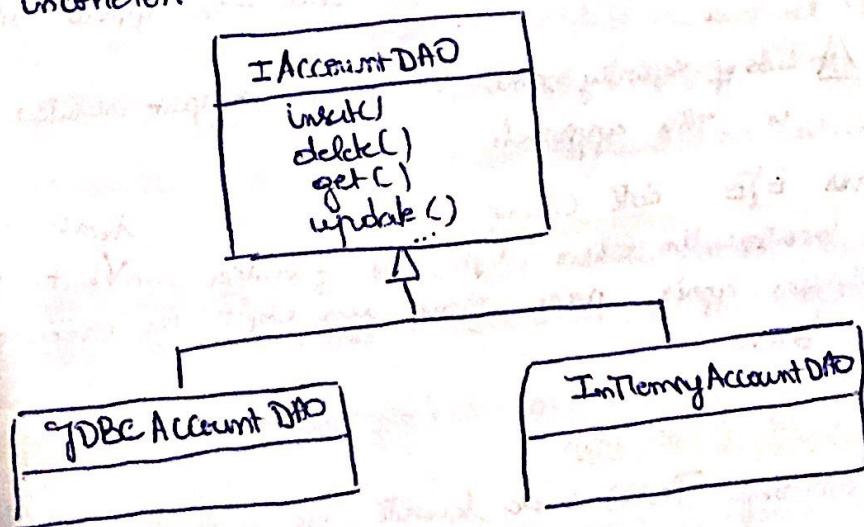
- \* **Echin Facade:** Pour réduire le nombre de proxy distants, coordonner les interactions avec plusieurs beans et réduire le couplage avec les composants EJB, on utilise une classe Facade de session qui expose au client les fonctionnalités offertes par les composants EJB et simplifie l'interface avec les beans
  - Il faut implémenter un workflow
  - Interface plus simple et uniforme
  - Traits d'appels distants
  - Sécurité et gestion des transactions centralisées



- \* **Message Facade:** Faire appeler à un ensemble d'opérations EJB sans que ce soit bloquant à l'aide d'un API de messagerie (JMS) à travers une façade

## • Design Patterns de Persistance:

- \* **Data Access Object:** Pour remédier à:
  - Complexité de l'accès aux données entre des données en memory, JDBC...
  - Accès aux données et la logique applicative donnent ensemble un code incohérent



## Cloud Native Principles

- ↳ an approach to building and running apps that exploits the advantages of the cloud
- To be cloud Native: abstracting many layers of infrastructure (network, service, OS...), allowing them to be defined in code
- Cloud Native Principles:
  - \* **Runtime Confinement:** Every container must declare its resource requirements and pass that information to the platform so that an application running on the cloud must confine to the requirements (CPU - Memory - Disk - Network - Auto-scaling...)
  - \* **Self Containment principle:** A container must contain everything that it needs at build time. (e.g.: libraries, dependencies...)
    - ↳ Shared Resources (e.g.: DB) must be contained separately
  - \* **Process Disposability:** A container needs to be ready to be replaced by another container instance at any point
    - Create small containers that start up quickly
  - \* **Image Immutability:** A new container image is built and reused across all environments if the containerized app has changed.
    - Prevent the creation similar images for ≠ environments
  - \* **Lifecycle Conformance:** the events coming from the managing platform are intended to help you manage the lifecycle of your container
  - \* **High Observability:** Containers are treated like black boxes.
    - They must provide APIs so that the runtime environment can observe the container's health and act accordingly
  - \* **Single Concern:** Every container must address a single concern and do it well.
    - ↳ A containerized service may address multiple concerns through design patterns like: sidecar, init containers... to combine container into a single deployable unit
- Good practice principles: for quality SW (cloud or not):
  - KISS: Keep it Simple, Stupid
  - DRY: Don't repeat yourself
  - YAGNI: You aren't going to need it
  - SOC: Separation of concerns.