# Architecture d’un module avec Domain Driven Disign

**Présentation du DDD**

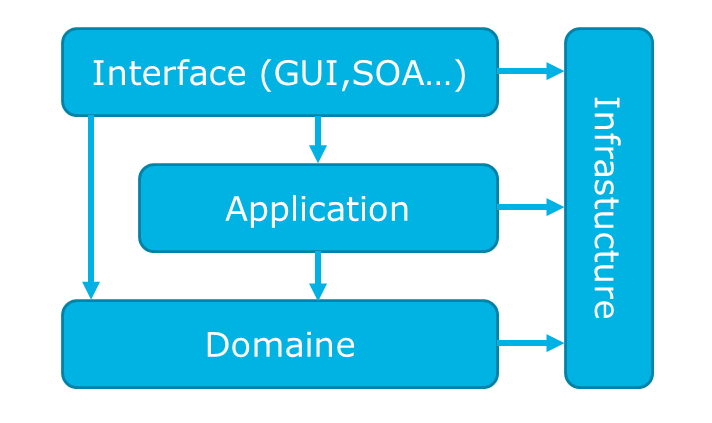
Le Domain Driven Disign n’est ni une méthode ni une technologie. **C’est est une manière de penser la conception autour du code, de collaborer et de communiquer avec les experts fonctionnels.**

Il a été introduit par Eric Evans en 2003. Il a construit cette approche de conception suite à des retours d’expérience. Le DDD est centré sur le métier de l’application et le code source qui l’implémente.

En effet, la conception est conduite par un modèle. Ce modèle est en partie constitué d’un langage de communication commun aux experts fonctionnels et aux équipes de développement appelé *Ubiquitous.*

Techniquement, C’est un ensemble de concepts et de design patterns. Il permet :

* Isolation de la couche domaine avec les concepts d’inversion de contrôle et d’injection de dépendances.
* Découpler toutes les couches d’infrastructure de la couche du domaine
* Utilisation de nombreux patterns d’architecture (MVC, CQRS…)
* Utilisation d’ORM

**

*Couche Domain :*

C’est le cœur du logiciel et la base de discussion avec un expert métier. Elle contient tous les concepts métier « purs » et les entités associées. En effet, elle ne doit pas être contaminée avec des problématiques issues d’autres couches et doit être indépendante de toute source de données ou technologie d’accès. De plus, toutes les dépendances doivent utiliser des abstractions ou des interfaces. On utilise des conteneurs IoC (comme CDI pour java EE) pour proposer des implémentations interchangeables En fin, toutes les entités (au sens ORM) sont présentes dans cette couche.

*Couche Application :*

Elle contient toute la plomberie de coordination entre les différentes couches. C’est la chef d’orchestre des uses cases métiers (synchrone ou asynchrone).En effet, Elle ne contient pas de métier mais elle gère le cycle de vie de l’entité métier. De plus, elle peut envoyer des Events de niveau applicatif et peut contenir des EventListeners de niveau domaine. En fin, elle contient et implémente tous les services applicatifs.

*Couche Infrastructure :*

C’est une commune aux différentes couches. Elle contient tous ce qui existe indépendamment de l’application : les librairies externes, le « Database Engine » ou l’entité manager pour l’ORM attaché. Et le plus important, l’implémenter des Repository définies dans la couche Domain.

*Couche Interface ou présentation :*

Elle se charge de l’interprétation, la validation et la transformation des données entrantes et aussi de la serialization des données sortantes : xml, html, json, etc.

*Concepts dans DDD :*

Bounded context : Context Borné défini par son langage (Ubiquitous language) : noms+verbes présents dans le code aussi

ValueObject : Objet dont l’état ne change pas (immutable), en général c’est une classe (pas une entité) réutilisable dans différentes Entités.

Entity : Objet dont l’état peut changer mais pas son identité (mutable), il est constitué d’attributs et de valueObjects (en JPA @Embedded). Dans DDD, une entité contient du code métier.

Aggregate : Objet (Entité) du domaine constitué d’entités, d’attributs et de valueObjects, il correspond à l’atomicité d’une transaction : intégrité de l’ensemble

Repository : il gère le cycle de vie des Entités. C’est comme un DAO sans en être un, voire après dans CQRS.

DTO (Data Transfert Object) : C’est un « subset » d’une entité métier. Il est plus léger que de transférer l’entité entière (un peu comme un InfoItem configuré pour une vue).

Shared Kernel (Noyau partagé) : il permet de partager des subsets communs aux domaines (référentiel commun)

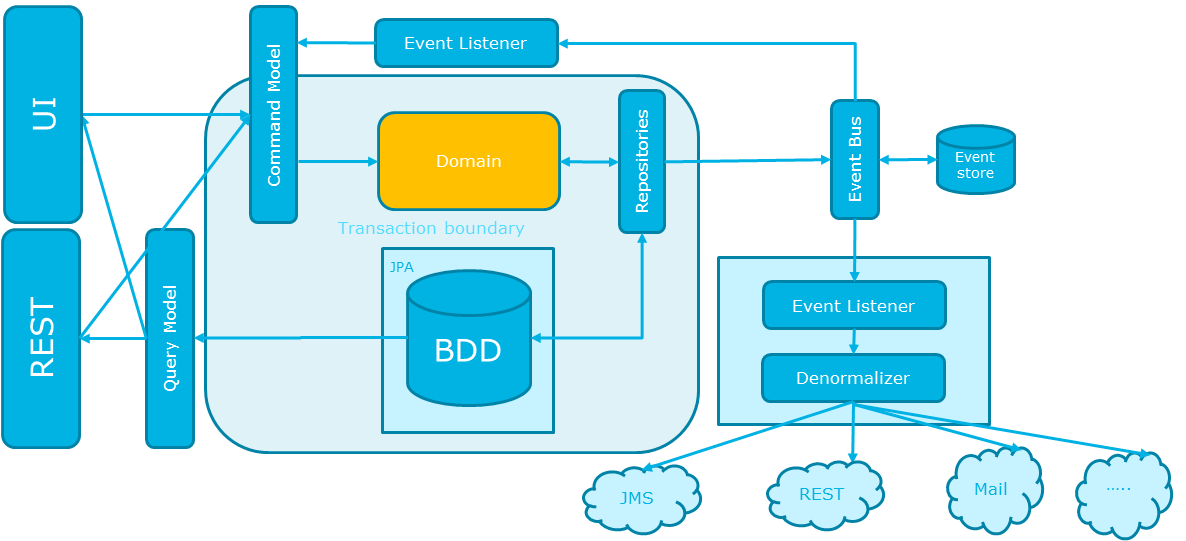
SAGA : c’est le process (events) manager entre bounded contexts.il permet par exemple de corréler différents events pour une prise de décision comme gestion du workflow d’events.

**Présentation de CQRS**

Le pattern CQRS (*Command Query Responsibility Seggregation*) repose sur un principe simple : la séparation, au sein d’une application, des composants de traitement métier de l’information (« command » / écriture) et de restitution de l’information (« query » / lecture).

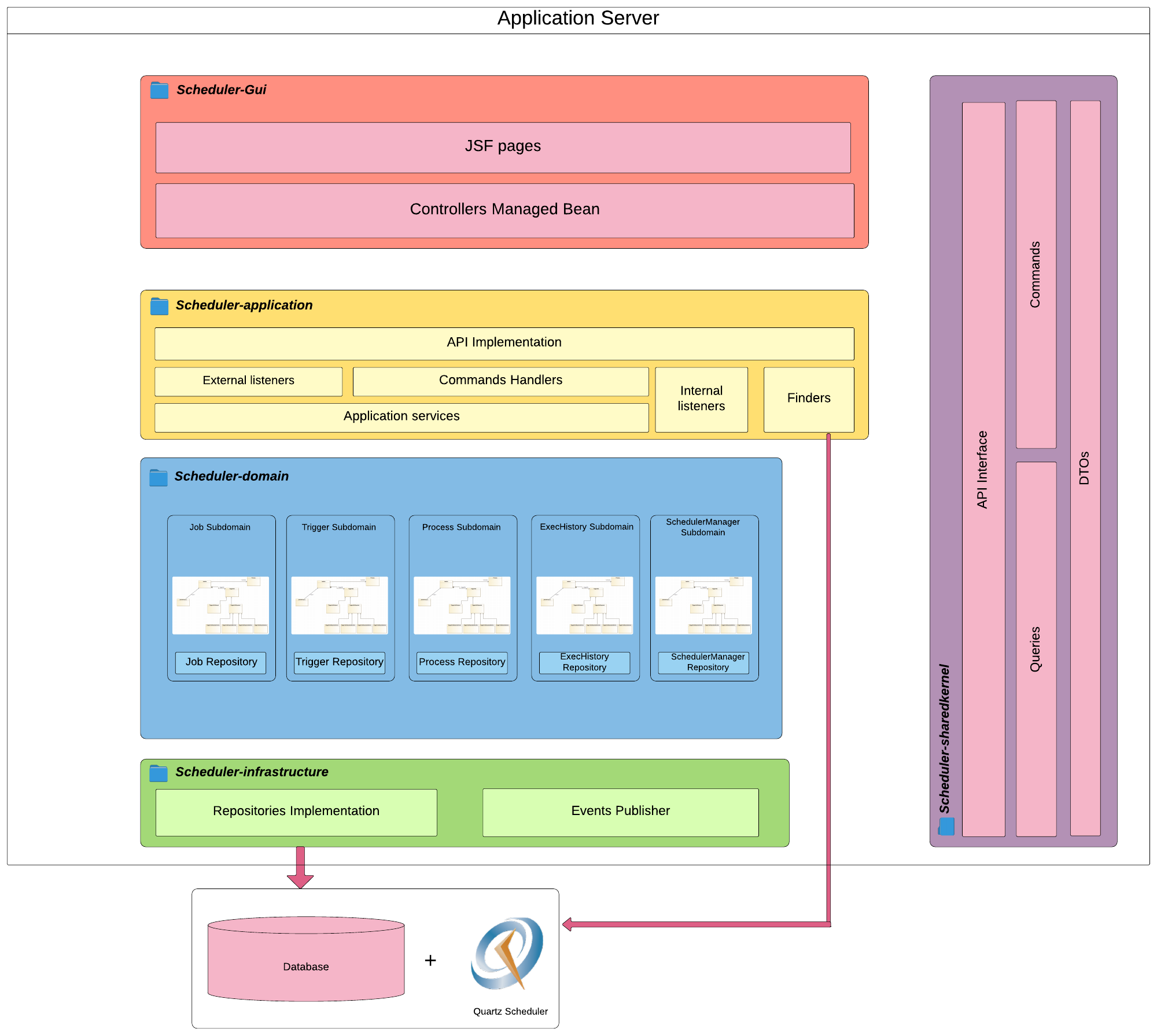
Ce seul principe fournit un cadre d’architecture extrêmement intéressant pour le développement d’applications, en levant un certain nombre de contraintes et en faisant apparaître de nouvelles opportunités.

**Architecture DDD avec CQRS**

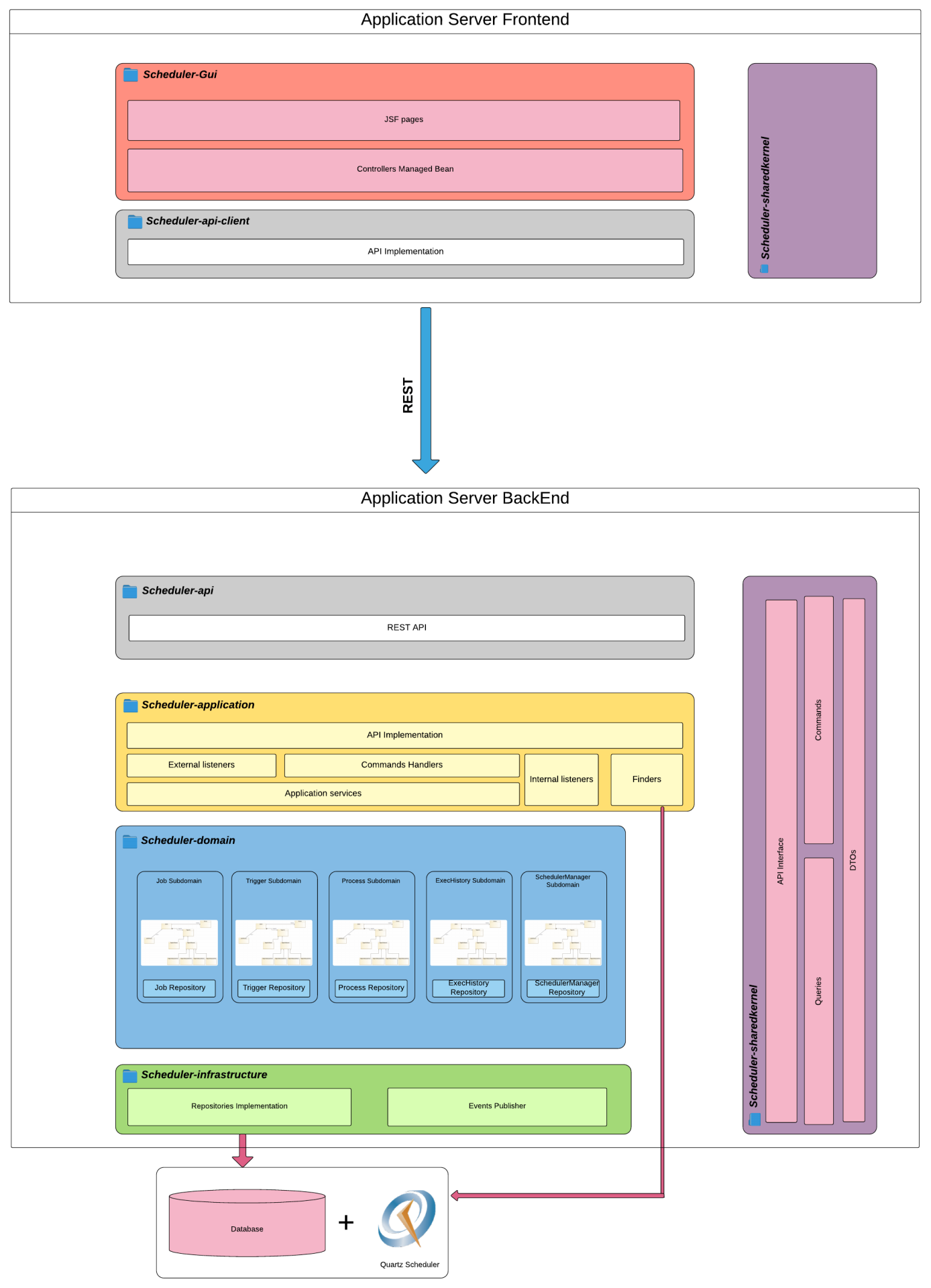


**Vues détaillées de l’architecture DDD en différents modes de déploiements : Exemple module scheduler NO**

**Mode 1 : Backend et frontEnd dans le meme serveur**



Mode 2 : backend et front end sur 2 serveur différents



Mode 3 : Backend avec api REST

