Domain-Driven Design (DDD)

Objectifs de cette section

Dans cette section, nous allons explorer l'approche **Domain-Driven Design (DDD)**, qui permet de structurer les applications en mettant l'accent sur le domaine métier. À la fin de cette section, vous serez capable de :

- Définir ce qu'est **le Domain-Driven Design (DDD)** et son importance.
- Comprendre les concepts clés du **modèle de domaine**.
- Identifier les **différences avec les autres architectures**.
- Implémenter une **application backend basée sur DDD en PHP et Node.js**.
- Analyser les **implications en infrastructure** et organiser le déploiement.

---

1. Qu'est-ce que le Domain-Driven Design (DDD) ?

Le **Domain-Driven Design (DDD)** est une approche qui vise à **organiser le code autour du domaine métier**, plutôt que de se focaliser sur la technologie ou l'architecture technique.

L'idée principale : Le logiciel doit refléter la **réalité métier** de l'organisation pour laquelle il est conçu.

Caractéristiques principales

- Priorité donnée à la **logique métier** plutôt qu'aux détails techniques.
- Organisation du code en **entités, agrégats, services et repositories**.
- Collaboration étroite entre **développeurs et experts métier**.
- **Utilisation d'un langage commun** appelé **Ubiquitous Language** entre toutes les parties prenantes.

Exemple concret : Dans une application de gestion de tâches :

- Une **Tâche** (`Task`) est une **Entité** qui possède des propriétés (`titre`, `description`, `statut`).
- Un **Utilisateur** (`User`) est également une **Entité** qui interagit avec la tâche.
- La **gestion du statut d'une tâche** peut être encapsulée dans un **Service Métier** (`TaskService`).
- Un **Repository** (`TaskRepository`) s'occupe de récupérer et stocker les tâches dans la base de données.

---

2. Différences entre DDD et autres architectures

Aspect	Architecture traditionnelle	Domain-Driven Design (DDD)
Approche	Orientée vers la base de données	Orientée vers le domaine métier
Organisation	Basée sur les couches techniques (MVC, n-tiers)	Basée sur des concepts métiers (Entités, Agrégats, Services)
Flexibilité	Difficulté à faire évoluer les règles métier	Plus modulaire et adaptable aux changements
Communication	Développeurs et experts métier travaillent séparément	Collaboration continue avec un langage commun

---

3. Concepts clés du Domain-Driven Design

DDD introduit plusieurs concepts importants pour structurer le code de manière logique et cohérente.

1. Entités

Des objets métier identifiables de manière unique.
Exemple : Une **tâche** (`Task`) a un `id`, un `titre` et une `description`.

2. Agrégats

Un ensemble d'entités qui doivent être modifiées ensemble pour garantir la cohérence des données.
Exemple : Un **Utilisateur** (`User`) possède plusieurs **Tâches** (`Task`), et toutes les modifications sont validées par `User`.

3. Services Métier

Contiennent la **logique métier** qui ne dépend pas d'une seule entité.
Exemple : Un **service de gestion des tâches** (`TaskService`) qui applique les règles métier (ex: **changer le statut d'une tâche**).

4. Repositories

Fournissent une **interface pour accéder aux données**, sans exposer directement la base de données.
Exemple : `TaskRepository` gère **l'enregistrement, la récupération et la suppression** des tâches.

5. Événements de Domaine

Permettent de **réagir aux actions métier** et de déclencher d'autres traitements.
Exemple : Lorsqu'une tâche est complétée, un événement `TaskCompleted` est généré pour notifier l'utilisateur.

---

4. Mise en place d'un backend basé sur DDD

Nous allons maintenant créer une **application de gestion de tâches** en **suivant les principes du DDD**.

Les technologies utilisées :

- **PHP + MySQL** pour la version PHP.
- **Node.js + PostgreSQL** pour la version JavaScript.
- **Découpage en Entités, Services et Repositories**.

---

5. Mise en pratique pour les administrateurs infrastructure

En plus du développement du backend, les administrateurs système devront analyser **l'hébergement et la gestion d'un backend basé sur DDD**.

5.1. Identification des composants techniques

Objectif : Analyser les besoins d'une architecture orientée domaine.

- Quels sont les **composants métier** principaux et leurs rôles respectifs ?
- Comment garantir que la logique métier reste **indépendante des technologies** utilisées ?
- Quels sont les besoins en **stockage et accès aux données** dans un projet DDD ?

Livrable attendu : Un schéma des **composants métier et de leur interaction avec les bases de données**.

5.2. Hébergement et organisation des services métier

Objectif : Déterminer **comment et où** héberger une architecture DDD.

- Comment organiser **les services métier et les bases de données** sur le serveur ?
- Quels outils permettent de **mettre en cache les résultats métier** pour éviter des requêtes répétitives ?
- Faut-il regrouper plusieurs services métiers sur un même serveur ou les séparer ?

Livrable attendu : Un plan détaillé sur l'hébergement des services métier et leur répartition.

5.3. Gestion des bases de données et cohérence transactionnelle

Objectif : Assurer **l'intégrité et la cohérence des données** dans un projet basé sur DDD.

- Comment garantir que toutes les modifications d'un **agrégat** sont bien enregistrées ensemble ?
- Quel rôle jouent les **événements de domaine** dans la persistance des données ?
- Comment éviter les conflits d'accès aux données dans un **environnement multi-utilisateurs** ?

Livrable attendu : Une stratégie de gestion transactionnelle des données et de prévention des conflits.

5.4. Sécurisation et monitoring d'une application DDD

Objectif : Assurer la **sécurité et la surveillance** d'un backend structuré en DDD.

- Comment protéger les **services métier** contre les accès non autorisés ?
- Quels outils peuvent être utilisés pour **monitorer la performance des services** ?
- Comment identifier et corriger les **erreurs métier avant qu'elles ne causent des dysfonctionnements** ?

Livrable attendu : Une check-list des **bonnes pratiques de sécurité et de surveillance d'un projet DDD**.