Find Your Course

TypeScript et Domain-Driven Design pour le développement d'API sur Node.js

AMAN Aria – BURY JAY

ASSIAKH Sami – TRAORE Moussa Seydou

Document de suivi

Travaux du 01/05/2024 au 28/02/2025

Établissement : École Supérieure Génie Informatique
Spécialisation : Ingénierie du Web
Mentor : M. SALVADOR

Contexte et objectifs

Le cours suivant a été créé dans le cadre du module Find Your Course (FYC), un projet des étudiants passant de 4e à 5e année à l'ESGI. Le but du FYC est de mettre par groupe quatre étudiants issus de la même ou de différentes spécialisations à travailler ensemble pour concevoir un cours de niveau master avec une thématique libre dans le domaine de l'informatique. Le module FYC permet aux étudiants de s'investir profondément dans un sujet qui n'est pas appris à l'école, tout en développant des compétences à la fois académiques et professionnelles.

La création de ce cours a plusieurs objectifs, le premier étant d'apprendre à effectuer une recherche approfondie sur un sujet afin de produire un support pédagogique. Ce support entièrement digitalisé sera hébergé sur la plateforme Moodle. En faisant ce projet, les étudiants acquerront des connaissances polyvalentes comme la capacité de veille technologique, la vulgarisation et la recherche indépendante. Pour faire cette recherche, il faut être attentif aux innovations et aux tendances du secteur, favorisant une connaissance profonde du sujet pour pouvoir l'expliquer à une tierce personne.

Le deuxième objectif de cette initiative est d'augmenter la visibilité professionnelle des étudiants. Dans un premier temps, la plateforme centralisée de l'ESGI va offrir un espace pour diffuser et valoriser leur travail auprès de la communauté. Dans un deuxième temps, des professionnels de l'informatique seront invités aux soutenances, permettant aux étudiants de réseauter et d'échanger des informations et des connaissances avec des membres du monde professionnel.

Enfin, le projet FYC encourage la création de contenus riches et variés sous différentes formes ; l'écriture, les vidéos, les graphiques et schémas, et autres supports servent à exposer les étudiants à une approche multimédia dans la création de leur cours en les rendant plus engageants et accessibles.

Alors qu'il s'agit d'un projet indépendant pour les étudiants, ils ne seront pas seuls. Chaque groupe est accompagné par un mentor, qui est un professionnel dans l'informatique et qui guidera les étudiants tout au long du projet. Son rôle est essentiel dans l'affinement du sujet choisi, la délimitation des problématiques et l'élaboration du contenu pédagogique. De plus, sa présence permettra aux étudiants de faire des points réguliers d'avancement et de demander des conseils concernant la recherche, la rédaction et la présentation du cours.

Ainsi, ce document a pour vocation d'organiser nos idées, d'archiver la documentation que nous aurons recueillie et de mettre en ordre nos réflexions. Il servira de base pour la structuration de notre travail en groupe et garantira une cohérence dans notre production finale.

Sommaire

Introduction

Avant toute introduction au cours, nous allons, dans un premier temps, faire un test de positionnement disponible sur notre plateforme Moodle, dans la section dédiée à la séance du 9 octobre 2024.

Ce test vise à avoir une vue d'ensemble sur les bases que possèdent les étudiants sur le sujet dans son ensemble, en vue d'améliorer l'enseignement en mettant un accent ou non sur une des parties du cours.

A. Contexte et importance

Dans le monde de développement aujourd'hui, les applications web et les backend deviennent de plus en plus complexes, et il existe de plus en plus de langages et d'outils pour simplifier la démarche. Parmi ces outils, TypeScript sort du lot comme incontournable pour les développeurs. Sur-ensemble de JavaScript, TypeScript offre plusieurs bénéfices aux développeurs, notamment en termes de sécurité et de maintenabilité.

Un des avantages principaux du TypeScript est le typage statique. Cette forme de typage va associer une variable à un type, comme un integer, un string ou un boolean. En faisant l'association d'un type à une valeur dès l'écriture d'une fonction, les erreurs de type seront identifiées avant l'exécution du code. Si les erreurs sont trouvées avant l'exécution, il est possible de corriger le problème plus tôt dans le développement et de diminuer les problèmes en production, créant une application plus fiable. Prenons comme exemple un projet collaboratif avec plusieurs développeurs : L'utilisation explicite des types assure une compréhension claire du comportement attendu des fonctions. En sachant à quoi s'attendre d'une fonction spécifique, les développeurs minimisent les erreurs potentielles en éliminant les incohérences liées aux types implicites. Dans une fonctionnalité importante d'une application où une erreur peut avoir des répercussions graves, cette mesure ajoute une couche importante à la lutte contre les erreurs.

En termes de **maintenabilité**, TypeScript contient plusieurs avantages qui facilitent la gestion de projet. Les fonctionnalités comme les interfaces, les classes et les types personnalisés permettent de structurer le code de manière claire et cohérente. Cette structuration devient indispensable lorsqu'une application se pose sur des bases de code volumineuses ou évolutives. Par exemple, dans une application comportant des centaines de modules, le typage aide à éviter les conflits, en rendant explicites les attentes en termes d'entrée et de sortie de chaque composant.

De plus, TypeScript connaît une adoption importante dans le monde professionnel. Des grandes entreprises comme Microsoft, Google, Airbnb et Slack utilisent TypeScript dans leurs projets. Cela reflète non seulement la maturité de l'outil, mais également sa pertinence dans un contexte moderne, où les besoins évoluent rapidement et la flexibilité d'un outil est primordiale. La compatibilité de TypeScript avec d'autres frameworks comme Angular, React et Vue.js en fait un choix naturel pour les développeurs front-end et full-stack.

Enfin, TypeScript est un langage qui est régulièrement mis à jour, ce qui nous permet de suivre les dernières avancées technologiques en assurant une **rétrocompatibilité** avec les anciennes versions de JavaScript. Avec ce rythme d'avancement, TypeScript trouve un

équilibre entre innovation et stabilité. Son adoption ne se limite pas à la résolution des problèmes actuels, mais prépare également les développeurs aux défis futurs.

À mesure que les systèmes logiciels deviennent de plus en plus complexes, les approches traditionnelles de conception atteignent leurs limites. Pour faire face à ces enjeux, le **Domain-Driven Design (DDD)** apparaît ainsi comme une approche méthodologique efficace. Introduit par Eric Evans en 2003, le DDD met l'accent sur la compréhension et la modélisation approfondies du **domaine métier**, qui est le cœur de tout système logiciel.

L'un des principaux objectifs du DDD est de réduire la **complexité cognitive**. Dans des projets où les règles métier et les interactions entre composants sont nombreuses, le DDD aide à diviser le système en sous-domaines clairs et indépendants, appelés **bounded contexts**. Ces contextes délimitent des espaces de responsabilité, permettant ainsi aux équipes de travailler sur des sections spécifiques du projet sans interférer les unes avec les autres. Cela améliore la scalabilité organisationnelle et technique.

Un autre avantage clé du DDD est qu'il met l'accent sur une **langue commune** (ubiquitous language) entre les développeurs et les experts métier. Avec l'utilisation des termes et des concepts compréhensibles par tous les intervenants, il favorise une meilleure collaboration et réduit les malentendus. Cette clarté de communication est cruciale dans des environnements où les exigences changent fréquemment, car elle permet une adaptation rapide et minimise le risque d'erreurs.

Dans le contexte des **systèmes scalables**, le DDD brille par son approche modulaire. Les concepts comme les agrégats, les objets de valeur et les entités permettent de concevoir des modèles résistants aux changements, tout en optimisant les performances. Par exemple, dans une application e-commerce, l'utilisation d'agrégats permet de regrouper les données relatives aux commandes, aux clients et aux paiements dans des unités logiques bien définies. Cela réduit les dépendances entre les composants et facilite la gestion de la montée en charge.

Enfin, le DDD s'intègre particulièrement bien dans les environnements modernes, où les architectures orientées microservices sont devenues courantes. Chaque microservice peut être conçu pour gérer un bounded context spécifique, ce qui simplifie la maintenance et améliore la résilience globale du système.

En résumé, le Domain-Driven Design est bien plus qu'une méthodologie ; c'est une philosophie de conception centrée sur la création de systèmes qui reflètent fidèlement les besoins métier tout en restant flexibles et évolutifs. Sa pertinence dans le développement moderne réside dans sa capacité à transformer des idées complexes en solutions concrètes, adaptées aux réalités techniques et organisationnelles.

Ces deux piliers, TypeScript et le Domain-Driven Design, constituent des outils complémentaires. TypeScript garantit une base technique solide, tandis que le DDD fournit une approche méthodologique pour concevoir des systèmes qui répondent aux besoins métier les plus exigeants. Cette combinaison est essentielle pour le développement d'API robustes et évolutives. Ce que nous explorerons en détail dans les sections suivantes.

B. Objectifs du cours

Ce cours est organisé en quatre grandes sections qui explorent la synergie entre **TypeScript**, **Domain-Driven Design (DDD)** et leur application dans le **développement d'API avec Node.js**. L'objectif est de fournir une compréhension complète de ces technologies, tout en montrant comment les intégrer pour créer des solutions robustes, maintenables et adaptées à des systèmes complexes.

Dans la première section, **présentation de TypeScript**, nous examinerons l'évolution de ce langage, depuis son apparition jusqu'à son adoption massive dans l'industrie. Nous aborderons également ses principaux avantages, notamment en termes de sécurité et de maintenabilité du code, sans oublier ses inconvénients, comme la courbe d'apprentissage initiale. Enfin, nous détaillerons les fonctionnalités essentielles de TypeScript. Cette section inclura, parmi d'autres sujets : les annotations de types, les interfaces, les generics et leur rôle dans la gestion des projets à grande échelle. Cette section posera les bases techniques nécessaires pour le reste du cours.

La deuxième section, **Domain-Driven Design**, se concentre sur les concepts clés du DDD, une approche méthodologique qui place le domaine métier au centre du processus de développement. Nous détaillerons les notions fondamentales telles que les bounded contexts, les agrégats et les objets de valeur, avant d'explorer leur application pratique dans la conception d'API. Pour illustrer ces principes, des études de cas et des exemples concrets seront présentés, vous permettant d'ajouter aux idées théoriques avec une mise en œuvre pratique.

Puis, la troisième section, **Intégration du DDD avec TypeScript et Node.js**, traitera la manière dont ces outils peuvent être combinés pour maximiser leur efficacité. Nous décrirons les meilleures pratiques pour appliquer le DDD dans un environnement TypeScript/Node.js et discuterons des outils et frameworks disponibles, comme NestJS et TypeORM, qui facilitent cette intégration. La section culminera par une démonstration pratique, où nous construirons une API en suivant les principes du DDD.

Enfin, la dernière section s'attarde sur l'application du DDD dans un cadre de gestion de projet. Elle aborde la planification initiale, la construction des équipes et la gestion des différentes phases du projet. Des exemples d'études de cas réels, incluant des succès et des échecs, sont également analysés. Cette section met l'accent sur l'alignement des objectifs métiers et techniques et l'utilisation d'approches structurées pour garantir la cohérence et la qualité des résultats.

En résumé, ce cours vise à équiper le lecteur de connaissances approfondies et de compétences pratiques pour concevoir des APIs robustes et évolutives. Il s'adresse aux développeurs souhaitant combiner théorie et pratique pour produire des solutions modernes, fiables et adaptées aux besoins métier.

Nous recommandons fortement d'avoir les prérequis suivants avant de suivre ce cours. Dans le cas où il vous manque quelques compétences, surtout en Typescript et Node.js, nous ferons une courte mise à niveau dans la prochaine section.

1. Connaissances fondamentales en programmation

- **Langages de base** : Une solide compréhension de JavaScript (ES6+) est indispensable, y compris :
    - Les fonctions, objets et classes.

    - Les concepts asynchrones : *promises*, async/await.
    - La manipulation du DOM et des structures de données courantes (tableaux, objets, map, set).
- Familiarité avec les paradigmes de programmation fonctionnelle et orientée objet.

2. Notions de base en développement d'API avec Node.js

- Compréhension des bases de Node.js, y compris :
    - Gestion des modules (require/import, exports/export).
    - Utilisation de npm ou yarn pour gérer les dépendances.
    - Conception de serveurs web simples avec des frameworks comme Express.js.
    - Utilisation des middlewares pour gérer les requêtes et réponses HTTP.

3. Bases en TypeScript

- Connaissances introductives de TypeScript, telles que :
    - Déclaration et utilisation de types simples (string, number, boolean, etc.).
    - Création et utilisation des interfaces et types personnalisés.
    - Concepts d'inférences de type et de contrôle des erreurs à la compilation.
    - Utilisation basique de classes et de modificateurs de visibilité (public, private, etc.).

4. Introduction aux concepts d'architecture logicielle

- Familiarité avec les principes de base tels que :
    - Modèles MVC (Model-View-Controller).
    - Couplage faible et modularité.
    - Concepts de services et de couches d'abstraction.
- Compréhension générale de ce qu'est une **API REST** (endpoints, stateless communication, etc.).

I. Présentation de Typescript

A. Historique et évolution

TypeScript est un langage de programmation open-source créé en 2012 par Microsoft et Anders Hejlsberg, l'architecte principal du langage C#. Son objectif initial était de combler les lacunes de JavaScript pour le développement d'applications complexes, en particulier dans des contextes industriels où la maintenance et la lisibilité du code sont critiques.

Le but de ce développement était d'améliorer et de sécuriser la production du code Javascript en évitant de rompre la compatibilité avec la norme **ECMAScript** et son écosystème. Pour faire cela, un compilateur a été développé pour transformer le Javascript typé en code ECMAScript 5 vanille, faisant de Typescript une surcouche à Javascript plutôt qu'un langage indépendant. La transformation de TypeScript en Javascript permet une compatibilité avec tous les navigateurs et environnements.

Avec les années, TypeScript a gagné en popularité grâce à son adoption par des frameworks. L'implémentation de TypeScript dans la version 2 d'**Angular** dès sa version 2 a notamment

joué un rôle clé dans sa démocratisation. La communauté a ensuite renforcé cette adoption en intégrant TypeScript dans d'autres projets phares, comme **React**, **Vue** et des outils de backend comme **NestJS**. Aujourd'hui, TypeScript est considéré comme l'un des langages essentiels pour le développement web moderne, avec une adoption croissante dans les entreprises pour des projets de grande envergure.

B. Avantages et inconvénients

1. Avantages

1. **Typage statique robuste**
   TypeScript permet de définir explicitement les types des variables, des fonctions et des objets, ce qui aide à détecter les erreurs dès la compilation. Cela évite des bogues courants liés à la nature dynamique de JavaScript. Exemple :

```
let age: number = 30;
age = "trente"; // Erreur détectée à la compilation
```

2. **Réduction des erreurs en production**
   Grâce au typage statique, les erreurs de type, souvent sources de crashs en production, sont éliminées lors de la phase de développement. Cela augmente la fiabilité du code.
3. **Autocomplétion et documentation améliorées**
   TypeScript enrichit l'expérience de développement en offrant une autocomplétion précise, une meilleure navigation dans le code et une documentation intégrée, notamment dans les IDE comme **Visual Studio Code**.
4. **Support des dernières fonctionnalités JavaScript**
   TypeScript compile vers du JavaScript standard (ES5 ou ES6), ce qui garantit sa compatibilité avec tous les navigateurs. Il intègre également les dernières fonctionnalités JavaScript avant leur adoption officielle, ce qui permet aux développeurs d'utiliser des outils modernes en toute sécurité.
5. **Interopérabilité avec JavaScript**
   TypeScript peut cohabiter avec du JavaScript classique, ce qui permet une transition progressive dans les projets existants. On peut commencer par ajouter des annotations de types aux fichiers .js avant de passer complètement à .ts.
6. **Écosystème riche et communautaire**
   Les fichiers de types (par exemple, @types/lodash) permettent une utilisation typée de nombreuses bibliothèques JavaScript, même celles qui n'ont pas été conçues avec TypeScript à l'origine. Cela offre une flexibilité maximale.

2. Inconvénients

1. **Complexité accrue**
   Pour des projets simples ou des prototypes rapides, l'ajout de TypeScript peut être perçu comme un surinvestissement. La mise en place de typages complexes, comme les *generics* ou les unions, peut alourdir le développement.

2. **Configuration initiale**
   Le fichier tsconfig.json permet de personnaliser le comportement de TypeScript, mais sa configuration peut être intimidante pour les nouveaux utilisateurs. Il exige une bonne compréhension des options disponibles, comme le ciblage d'une version de JavaScript ou l'inclusion/exclusion de fichiers.
3. **Temps de compilation**
   Contrairement à JavaScript, TypeScript nécessite une étape de compilation. Dans les projets de grande taille, cela peut ralentir le cycle de développement, bien que des outils comme **esbuild** ou **Vite** atténuent ce problème.
4. **Apprentissage des concepts avancés**
   Les développeurs doivent apprendre des notions nouvelles, telles que les types avancés, les décorateurs ou encore les interfaces. Cela peut rallonger le temps nécessaire pour maîtriser TypeScript.

C. Principales fonctionnalités

TypeScript est un langage de programmation fortement typé qui étend JavaScript en introduisant des fonctionnalités avancées pour la création de projets robustes et maintenables. Ces fonctionnalités sont particulièrement adaptées aux projets complexes, ce qui en fait un outil incontournable pour les développeurs. Explorons ces fonctionnalités en détail.

1. Typage statique et inférence des types

Le typage statique est l'une des caractéristiques fondamentales de TypeScript. Contrairement à JavaScript, où les types sont dynamiques et peuvent changer au cours de l'exécution, TypeScript impose des types fixes qui sont vérifiés à la compilation. Cela permet de détecter les erreurs tôt dans le cycle de développement, améliorant ainsi la fiabilité du code.

Exemple d'inférence des types :

```
let name = "Alice"; // Type 'string' automatiquement déduit
name = 42; // Erreur : Type 'number' n'est pas assignable au type 'string'.
```

L'inférence de type réduit le besoin d'annotations explicites tout en offrant une vérification des types robuste. Cela permet d'écrire un code concis tout en minimisant les erreurs.

Avantages pour les développeurs avancés :

- Réduction des bogues liés aux erreurs de typage.
- Documentation implicite : les types servent de contrat pour les développeurs collaborant sur un projet.

2. Interfaces et types personnalisés

Les interfaces et les types sont des outils puissants pour modéliser des structures complexes dans TypeScript. Les interfaces permettent de définir des contrats pour les objets, tandis que les types personnalisés permettent de combiner ou de restreindre des structures de données.

Exemple d'interface :

```
interface User {
  id: number;
  name: string;
  email?: string; // Le point d'interrogation indique que le champ est optionnel
}

const user: User = { id: 1, name: "Alice" }; // Valide
```

Types avancés :

Les types personnalisés peuvent combiner plusieurs structures à l'aide des opérateurs | (union) et & (intersection).

```
type ApiResponse =
  | { success: true; data: any }
  | { success: false; error: string };
```

Applications pratiques :

- Modélisation de réponses d'API.
- Création de contrats stricts pour les bibliothèques et les frameworks.

3. Classes et modificateurs d'accès

TypeScript enrichit JavaScript avec des fonctionnalités orientées objet. Les classes permettent de structurer le code, tandis que les modificateurs d'accès comme private, public et protected apportent un contrôle précis sur la visibilité des propriétés et méthodes.

Exemple de classe avec modificateurs d'accès :

```
class Person {
  private name: string;

  constructor(name: string) {
    this.name = name;
  }

  greet(): string {
    return `Hello, ${this.name}`;
  }
}

const person = new Person("Alice");
// person.name = "Bob"; // Erreur : 'name' est privé
```

Avantages pour des projets complexes :

- Encapsulation : protège les données internes des modifications externes non contrôlées.
- Maintenance facilitée grâce à une structure claire et à une hiérarchie définie.

4. Generics

Les generics permettent de créer des composants réutilisables tout en conservant une vérification stricte des types. Cela est particulièrement utile pour les structures de données et les algorithmes génériques.

Exemple avec une fonction générique :

```
function identity<T>(value: T): T {
  return value;
}

const num = identity<number>(42); // T est un 'number'
const str = identity<string>("Hello"); // T est un 'string'
```

Applications avancées :

- Création de bibliothèques ou API indépendantes des types spécifiques.
- Manipulation de collections comme les tableaux ou les maps, tout en garantissant l'intégrité des données.

5. Décorateurs

Les décorateurs sont une fonctionnalité puissante de TypeScript qui permet d'ajouter des métadonnées ou de modifier le comportement des classes, méthodes ou propriétés. Ils sont particulièrement utiles dans les frameworks comme Angular ou NestJS.

Exemple de décorateur simple :

```
function Log(target: any, key: string) {
  console.log(`Appel de la méthode : ${key}`);
}

class Calculator {
  @Log
  add(a: number, b: number): number {
    return a + b;
  }
}
```

Utilisation pratique :

- Définir des routes ou des services dans les frameworks backend.
- Ajouter des validations automatiques ou de la journalisation.

6. Modules et namespaces

La modularité est essentielle pour les projets à grande échelle. TypeScript prend en charge les modules modernes d'ES6, permettant de diviser le code en fichiers réutilisables et bien organisés.

Exemple de module :

```
// mathUtils.ts
export function add(a: number, b: number): number {
  return a + b;
}

// main.ts
import { add } from './mathUtils';
console.log(add(5, 10)); // 15
```

Avantages :

- Favorise la réutilisation du code.
- Simplifie la maintenance en réduisant les dépendances croisées.

7. Types avancés

TypeScript propose des outils avancés comme les types conditionnels, les mapped types et les unions, permettant de définir des structures complexes de manière concise.

Exemple : mapped types :

```
type ReadonlyUser = {
  readonly [K in keyof User]: User[K];
};
```

Applications pratiques :

- Transformer dynamiquement des types existants.
- Création de contrats adaptés à des besoins spécifiques (par exemple, un utilisateur en lecture seule).

8. Système de compilation flexible

Le fichier tsconfig.json offre un contrôle détaillé sur la configuration du compilateur TypeScript. Les développeurs peuvent cibler différentes versions de JavaScript (es5, es6), activer des vérifications strictes et gérer les modules.

Exemple de configuration :

```
{
  "compilerOptions": {
    "target": "es6",
    "module": "commonjs",
    "strict": true
  }
}
```

Avantages pour les développeurs :

- Adaptabilité aux environnements modernes.
- Flexibilité pour l'intégration dans des pipelines CI/CD.

II. Domain Driven Design

A. Concepts du DDD

Le Domain-Driven Design, ou conception pilotée par le domaine, est une méthodologie introduite par Eric Evans en 2003 dans son ouvrage *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Son objectif principal est de faciliter le développement de logiciels complexes en alignant la modélisation technique du système avec les besoins métiers. Contrairement à une approche purement technique, le DDD place le domaine métier au cœur des préoccupations des équipes techniques et non techniques.

L'idée centrale du DDD est de modéliser le code de manière à refléter le langage et les concepts métiers, permettant ainsi une meilleure collaboration entre développeurs et experts métier. Cette méthodologie repose sur plusieurs concepts clés, dont le langage ubiquitaire (Ubiquitous Language), les contextes bornés ou délimités (Bounded Contexts), les agrégats et les objets de valeur, qui seront explorés en détail ci-dessous.

1. Ubiquitous Language

L'Ubiquitous Language ou "langage ubiquitaire" est l'un des piliers fondamentaux du DDD. Il s'agit d'un vocabulaire commun et partagé entre les développeurs et les experts métier. Ce langage est utilisé pour décrire les règles, les interactions et les entités spécifiques au domaine, à la fois dans la communication entre les équipes et dans le code source lui-même.

L'objectif principal de ce langage est d'éliminer les ambiguïtés qui pourraient découler d'une interprétation erronée des termes métiers.

Par exemple, dans un système de gestion de prêts, un terme comme "prêt" peut signifier une action d'emprunter ou un contrat d'emprunt en cours. En définissant clairement ces distinctions dès le départ, le langage ubiquitaire permet d'assurer une compréhension commune et une modélisation rigoureuse.

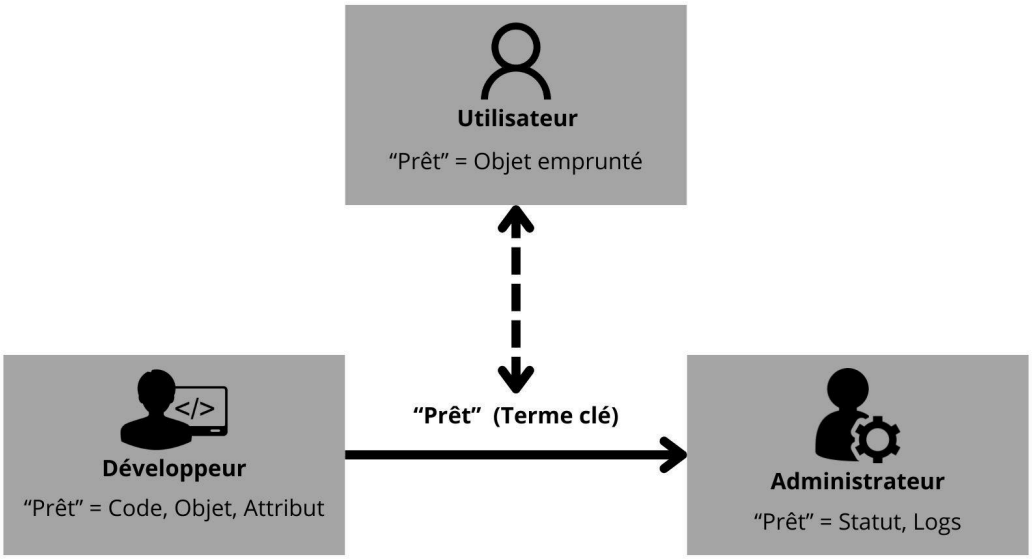

Schéma comparatif montrant l’interprétation d’un terme clé (ex. : "prêt") par différentes parties prenantes (développeur, utilisateur, administrateur).

2. Bounded Context

Un système logiciel complexe est rarement monolithique. Dans le cadre du DDD, le concept de contexte borné sert à diviser l’application en plusieurs domaines métiers distincts, appelés "bounded contexts". Chaque contexte est responsable d’un sous-ensemble cohérent de fonctionnalités et dispose de son propre langage ubiquitaire.

Cette séparation permet d’éviter les conflits de terminologie entre différents domaines. Par exemple, dans une plateforme e-commerce, un utilisateur peut être défini différemment selon les contextes. Dans le contexte de "Facturation", l’utilisateur est vu comme un client, tandis que dans le contexte "Catalogue produit", il pourrait être simplement un visiteur.

Caractéristiques principales :

- **Isolation des règles métiers** : chaque contexte encapsule ses propres règles et modèles de données.
- **Flexibilité et évolutivité** : les modifications dans un contexte n’affectent pas les autres.

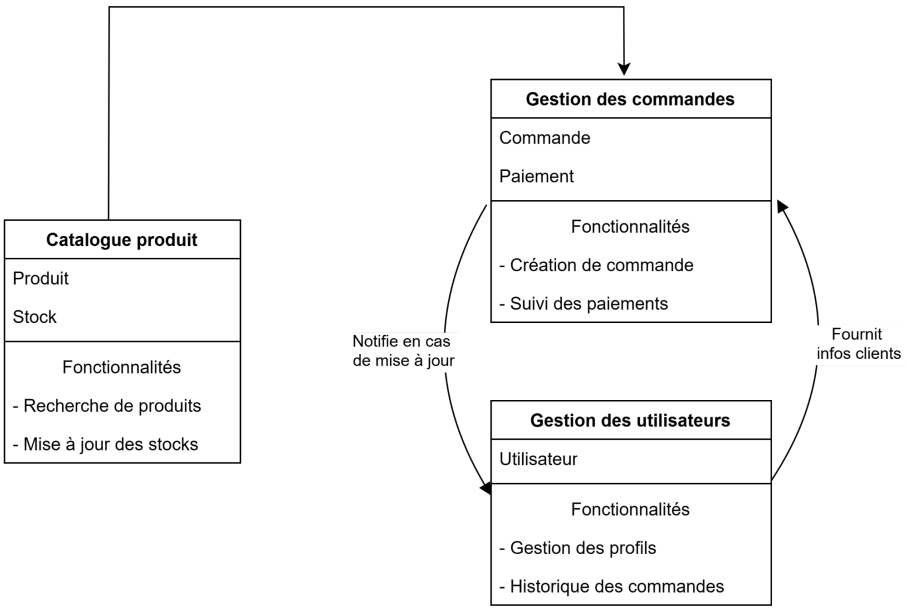

Diagramme illustrant une architecture divisée en plusieurs contextes bornés : Catalogue produit, gestion des commandes, gestion des utilisateurs

3. Agrégats et objets de valeur

Agrégats : les agrégats sont des regroupements d'objets liés qui forment une unité logique et cohérente dans le domaine. Chaque agrégat possède une entité racine (Aggregate Root), qui est le seul point d'entrée pour interagir avec les données de l'agrégat. Cette approche garantit que les règles métier définies dans un agrégat sont toujours respectées.

Exemple : dans un système de gestion de commandes, un agrégat "Commande" peut inclure des objets comme "Ligne de commande" et "Adresse de livraison". La racine de l'agrégat serait l'entité "Commande".

Objets de valeur : les objets de valeur sont des classes immuables qui encapsulent des données simples mais significatives pour le domaine. Contrairement aux entités, les objets de valeur n'ont pas d'identité propre. Leur égalité est basée sur leurs valeurs et non sur une identité unique.

Exemple : une adresse, composée de propriétés comme la rue, la ville et le code postal, peut être un objet de valeur partagé entre plusieurs agrégats.

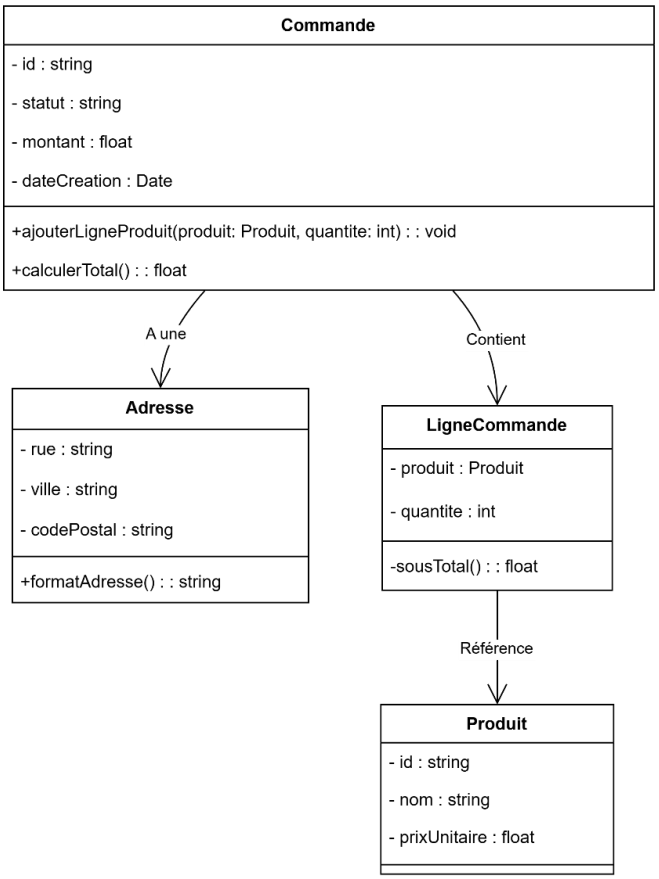

Schéma UML d'un agrégat et de ses relations avec des objets de valeur

Ces concepts clés constituent les fondations du Domain-Driven Design. En combinant le langage ubiquitaire, les contextes bornés et une gestion structurée des entités via les agrégats et les objets de valeur, le DDD propose une méthodologie puissante pour concevoir des systèmes logiciels alignés sur les besoins métiers. Ces outils stratégiques permettent non seulement de réduire la complexité des systèmes, mais aussi de garantir leur maintenabilité et leur évolutivité sur le long terme. Par la suite, nous verrons maintenant que le DDD améliore la conception et la maintenance des APIs.

B. Application du DDD pour le développement d'API

Le DDD offre une méthodologie particulièrement adaptée pour concevoir des APIs robustes, évolutives et alignées sur les besoins métier. Cette section explore les principaux aspects de l'application du DDD dans le contexte des APIs : modularité, gestion des erreurs, validation et documentation.

1. Modularité et découplage

L'un des principes fondamentaux du DDD est la division des responsabilités en contextes bornés (*bounded contexts*). Dans le cadre des APIs, cela signifie que chaque domaine métier peut être encapsulé dans un service ou microservice distinct.

Exemple pratique : dans une plateforme bancaire, on peut avoir :

- Un service dédié aux comptes clients, gérant la création et la gestion des comptes.
- Un autre service pour les transactions, chargé de traiter les paiements et les transferts.

Chaque service est autonome et dispose de ses propres modèles et règles métier. Ce découplage réduit les interdépendances entre les services, rendant le système plus résilient et évolutif.

2. Gestion des validations métier et des erreurs

En DDD, les agrégats jouent un rôle clé dans la validation des règles métier au sein des APIs. Cela garantit que toutes les règles métier sont appliquées de manière centralisée.

Prenons un système de prêt d'objets. Un agrégat "Prêt" pourrait vérifier :

1. Que l'objet est disponible.
2. Que l'utilisateur n'a pas dépassé le nombre maximum de prêts actifs.
3. Que la durée du prêt ne dépasse pas une limite de 30 jours.

Ces validations sont encapsulées dans l'agrégat et exposées via des endpoints d'API, comme :

- POST /loans pour initier un prêt.
- PUT /loans/:id/return pour retourner un objet.

3. Documentation naturelle grâce au ubiquitous language

Le concept de langage ubiquitaire (ubiquitous language) simplifie la documentation et améliore la collaboration entre développeurs et experts métier. En utilisant un vocabulaire commun dans les modèles et les APIs, il devient plus facile pour toutes les parties prenantes de comprendre le fonctionnement du système.

Exemple : dans une API e-commerce :

- Le terme "Commande" est utilisé uniformément pour désigner une transaction client.
- Les endpoints reflètent cette terminologie :
    - GET /orders pour récupérer les commandes.
    - POST /orders pour en créer une.

Cela réduit les malentendus et permet une transition plus fluide entre les exigences métier et leur implémentation technique.

4. Documentation naturelle grâce au ubiquitous language

Grâce au découplage et à la modularité, les APIs basées sur le DDD peuvent évoluer indépendamment. Par exemple, un service peut être mis à jour ou remplacé sans affecter les autres, à condition que ses contrats d'API restent stables.

Si un service de gestion des stocks nécessite une refonte pour inclure des fonctionnalités avancées, ses endpoints d'API peuvent rester inchangés, évitant ainsi des régressions pour les autres services consommateurs.

C. Études de cas et mise en œuvre

1. Étude de cas : Système de prêt et d'échange

L'implémentation d'un système de prêt et d'échange est un exemple concret où le Domain-Driven Design (DDD) peut être utilisé pour répondre aux exigences métier tout en garantissant la robustesse et la maintenabilité du code.

Définition des agrégats :

1. **Utilisateur :** représente les individus qui utilisent la plateforme.
    - **Attributs :** ID, nom, email, objets empruntés (liste des prêts actifs).
    - **Règle métier :** un utilisateur ne peut pas emprunter plus de 3 objets simultanément.
2. **Objet :** Représente les articles disponibles pour prêt ou échange.
    - **Attributs :** ID, nom, catégorie, disponibilité (booléen), propriétaire.
    - **Règle métier :** un objet ne peut être prêté que s'il est marqué comme disponible.
3. **Prêt :** décrit le processus d'emprunt.
    - **Attributs :** ID, objet (référence), emprunteur (référence), date de début, date de fin, statut (actif, retourné).
    - **Règle métier :** la durée maximale d'un prêt est de 30 jours.

Endpoints REST :

- POST /objects/{id}/loan : permet de créer un prêt.
    - **Conditions :** L'objet doit être disponible ; l'utilisateur ne doit pas dépasser la limite de 3 prêts actifs.
    - **Réponse :** détails du prêt ou message d'erreur si une règle est violée.
- PUT /loans/{id}/return : permet le retour d'un objet.
    - **Conditions :** le prêt doit être actif.
    - **Actions :** mise à jour de la disponibilité de l'objet et application des pénalités si le délai est dépassé.

2. Exemple TypeScript : Validation des règles métier

L'utilisation de TypeScript avec DDD permet de capturer les règles métier directement dans le code tout en renforçant la lisibilité et la maintenabilité.

Exemple de code pour l'entité Prêt :

```
class Loan {
    constructor(
        public id: string,
        public object: Object,
        public borrower: User,
        public startDate: Date,
        public dueDate: Date
    ) {
        if (this.dueDate.getTime() > this.startDate.getTime() + 30 * 24 * 60 * 60 * 1000) {
            throw new Error("La durée maximale d'un prêt est de 30 jours.");
        }
        if (!object.isAvailable) {
            throw new Error("L'objet n'est pas disponible.");
        }
        if (borrower.borrowedItems.length >= 3) {
            throw new Error("L'utilisateur a atteint la limite maximale de prêts.");
        }
    }
}
```

Description des points forts :

- Centralisation des règles métier : les validations sont directement implémentées dans l'entité Loan, garantissant une cohérence métier.
- Réutilisabilité : les agrégats peuvent être utilisés dans d'autres services ou modules sans modification.
- Sécurité : L'approche stricte de TypeScript empêche les erreurs de type ou les incohérences.

3. Étude approfondie : extension du système pour les notifications

Pour démontrer la scalabilité et l'évolutivité de ce système, envisageons l'ajout d'une fonctionnalité de notifications.

Objectif : notifier les utilisateurs lorsque :

- Un prêt arrive à échéance.
- Un objet qu'ils recherchent devient disponible.

Approche DDD :

- **Agrégat Notification :**
  - **Attributs :** ID, type (email, SMS), destinataire, message, statut (envoyé, en attente).
  - **Règle métier :** les notifications pour les échéances doivent être envoyées 2 jours avant la date limite du prêt.
- **Service d'envoi :**
  - Implémentation dans une couche dédiée, indépendante des autres services.
  - Utilisation de patrons comme les événements de domaine pour déclencher les notifications.

Exemple TypeScript pour une notification d'échéance :

```typescript
class NotificationService {
  sendDueDateReminder(loan: Loan): void {
    const now = new Date();
    const daysBeforeDue = (loan.dueDate.getTime() - now.getTime()) / (1000 * 60 * 60 * 24);

    if (daysBeforeDue <= 2 && loan.status === "active") {
      console.log(`Rappel envoyé à ${loan.borrower.email}: votre prêt arrive à échéance.`);
    }
  }
}
```

III. Intégration du DDD avec TypeScript et NodeJS

A. Meilleures pratiques

Maintenant que nous avons présenté le TypeScript, le DDD et ainsi que les principes d'application les plus importants. Nous allons maintenant voir comment intégrer tout cela au développement d'une API avec Node.js.

Pour ce faire, il est important de respecter certaines pratiques qui seront expliquées en prenant pour exemple une plateforme de vente de pièces électroniques. Nous nous concentrerons sur trois couches principales : le domaine, l'application et l'infrastructure.

1. Couche du domaine

La couche de domaine contient les entités, les objets de valeur et les racines d'agrégat qui modélisent les concepts clés du métier. Elle implémente les règles métier et garantit leur cohérence. Cette couche doit rester totalement indépendante de l'infrastructure pour éviter tout couplage et favoriser la réutilisabilité du code.

2. Couche applicative :

La couche applicative orchestre les cas d'utilisation en coordonnant les objets de domaine et en appliquant les règles métier définies. Elle sert de point d'entrée pour les opérations métier spécifiques et assure la séparation avec les détails techniques de l'infrastructure, comme la gestion des données ou les intégrations externes.

3. Couche d'infrastructure

La couche d'infrastructure gère les détails techniques tels que l'accès aux bases de données, les intégrations avec des services externes et l'utilisation de frameworks. Elle fournit des implémentations concrètes des interfaces définies dans les couches supérieures, tout en restant découplée du domaine et de l'application pour garantir une flexibilité et une évolutivité optimales.

Avantages de la séparation :

En séparant les différentes préoccupations de l'application, nous pouvons obtenir plusieurs avantages :

1. Modularité : chaque couche peut être développée et maintenue indépendamment, ce qui permet une collaboration et une réutilisation du code plus faciles.
2. Testabilité : avec des limites clairement définies entre les couches, il devient plus facile d'écrire des tests unitaires et de simuler des dépendances, garantissant ainsi l'exactitude de chaque composant.
3. Flexibilité : séparer les règles métier des détails de l'infrastructure permet d'apporter des modifications dans un domaine sans impacter les autres. Cela offre la flexibilité nécessaire pour s'adapter à l'évolution des besoins métier ou pour remplacer des composants d'infrastructure.

Pour conclure, en séparant les règles métier, les règles d'application et les problèmes d'infrastructure, nous pouvons créer des applications plus faciles à maintenir, plus évolutives et plus adaptables. La clé est de garder le domaine principal concentré, la couche d'application orchestrant et la couche d'infrastructure abstraite. L'application de DDD avec Node.js permet aux développeurs de créer des solutions logicielles qui s'alignent étroitement sur le domaine du problème et favorisent la durabilité à long terme.

B. Outils et Framework

L'intégration du Domain-Driven Design dans un projet TypeScript/Node.js peut être facilitée grâce à des outils et frameworks spécifiques. Ceux-ci permettent de structurer le code, de respecter les principes du DDD et d'assurer la maintenabilité et l'évolutivité de l'application. Voici un aperçu des principaux outils et frameworks recommandés :

1. NestJS : Un framework pour une architecture modulaire

NestJS est un framework Node.js progressif qui repose sur TypeScript. Il se distingue par son approche modulaire et son support intégré pour des architectures en couches, ce qui en fait un choix naturel pour appliquer les principes du DDD.

- **Points forts :**
  - Structure modulaire facilitant la séparation des contextes bornés.
  - Support des décorateurs TypeScript pour une définition explicite des dépendances et des modules.

    - Intégration facile avec des bibliothèques populaires comme TypeORM, MongoDB et CQRS.
    - Fournit des outils natifs pour les concepts du DDD comme les services, les modules et les intercepteurs.
- **Exemple d'utilisation :**
    - Les modules NestJS peuvent être utilisés pour modéliser des contextes bornés. Par exemple, un module Order peut encapsuler toute la logique métier relative aux commandes.

```
@Module({
  providers: [OrderService, OrderRepository],
  controllers: [OrderController],
})
export class OrderModule {}
```

2. TypeORM : Un ORM orienté objet pour les entités et agrégats

TypeORM est un ORM (Object Relational Mapper) qui facilite la manipulation des bases de données dans un style orienté objet. Il est idéal pour implémenter les entités, agrégats et objets de valeur dans le DDD.

- **Points forts :**
    - Mapping direct entre les entités de domaine et les tables de base de données.
    - Support pour les transactions, les relations, et les requêtes complexes.
    - Possibilité d'annoter les entités avec des décorateurs pour définir leurs propriétés et relations.
- **Exemple d'utilisation :**
    - Création d'une entité Order qui peut inclure des relations avec des objets de valeur comme une adresse.

```
@Entity()
export class Order {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  status: string;

  @OneToOne(() => Address, { cascade: true })
  @JoinColumn()
  deliveryAddress: Address;
}
```

3. CQRS (Command Query Responsibility Segregation)

Le pattern CQRS est particulièrement adapté pour les projets DDD. Il permet de séparer les opérations de lecture (query) des opérations d'écriture (command).

- **Points forts :**
    - Renforce l'isolation des règles métier en les associant aux commandes.
    - Permet une gestion optimisée des requêtes complexes, souvent nécessaires pour les vues spécifiques.

    - Facilement implémentable avec des frameworks comme NestJS, qui propose une bibliothèque CQRS intégrée.
- **Exemple d'utilisation :**
    - Implémentation d'une commande CreateOrderCommand et de son handler associé.

```
export class CreateOrderCommand {
  constructor(public readonly userId: string, public readonly items: OrderItem[]) {}
}

@Injectable()
export class CreateOrderHandler implements ICommandHandler<CreateOrderCommand> {
  async execute(command: CreateOrderCommand) {
    // Ici, on retrouve notre logique métier pour créer une commande
  }
}
```

4. Event Sourcing et Message Brokers

Dans des architectures complexes, l'utilisation de l'Event Sourcing et de message brokers comme Kafka ou RabbitMQ peut renforcer l'application des principes du DDD.

- **Event Sourcing :**
    - Enregistre chaque changement d'état sous forme d'événements immuables.
    - Permet une traçabilité complète des modifications dans le domaine.
- **Message Brokers :**
    - Facilitent la communication entre différents contextes bornés via des événements de domaine.
    - Par exemple, lorsqu'une commande est confirmée, un événement OrderConfirmed peut être publié pour informer d'autres services.

```
class OrderConfirmedEvent {
  constructor(public readonly orderId: string, public readonly userId: string) {}
}
```

5. Autres outils complémentaires

- **Jest** pour les tests unitaires : Garantit que les règles métier encapsulées dans les entités et agrégats fonctionnent correctement.
- **Swagger** pour la documentation : Génère automatiquement la documentation des APIs, renforçant l'utilisation du langage omniprésent.
- **Docker et Kubernetes** : Pour isoler les services correspondant à chaque contexte borné et faciliter leur déploiement.

C. Démonstration pratique

Cette partie détaillera un cas pratique complet d'application des concepts de Domain-Driven Design (DDD) pour construire une API RESTful avec **TypeScript** et **Node.js**. Nous suivrons les étapes essentielles en structurant notre projet autour des couches de **Domaine**, **Application** et **Infrastructure**. L'exemple choisi porte sur une plateforme de gestion de bibliothèques permettant d'emprunter et de retourner des livres.

Dans cette partie, l'objectif est de montrer l'intégration pratique du Domain-Driven Design (DDD) avec TypeScript et Node.js dans un projet concret. Cependant, pour favoriser un apprentissage plus interactif et dynamique, la démonstration complète a été réalisée sous forme de vidéo explicative disponible sur Moodle.

IV. Gestion d'un projet avec le Domain-Driven Design

La gestion de projet consiste à planifier, organiser et coordonner les ressources et les tâches nécessaires pour atteindre des objectifs spécifiques dans un délai défini. Elle est essentielle pour garantir le respect des échéances, optimiser les coûts et minimiser les risques.

Dans le développement d'applications, une gestion de projet efficace permet d'assurer la cohérence entre les besoins des parties prenantes et la réalisation technique et favorise la collaboration et l'adaptabilité face aux imprévus. En adoptant une approche structurée, on peut maximiser les chances de succès d'un projet et répondre aux exigences de qualité.

Comme vous le savez probablement, la gestion de projet joue un rôle central dans le développement d'une application ou un logiciel. Elle est souvent étroitement liée à la méthodologie de développement adoptée, car ces deux éléments doivent être alignés pour garantir les meilleurs résultats.

Dans ce chapitre, nous explorerons les étapes concrètes à suivre pour gérer un projet en intégrant le Domain-Driven Design (DDD), depuis la phase de conception jusqu'à la livraison finale.

A. Planification initiale

1. Définition des objectifs métiers et techniques

Le métier constitue le point d'entrée fondamental dans une approche DDD, car il est placé au centre de la conception. L'objectif principal de cette première phase est d'améliorer la connaissance métier en recueillant autant d'informations que possible afin de les partager. Cela permettra de mettre en lumière les concepts fondamentaux liés à leur expertise, et les expliciter.

- **Collecte des informations métiers :**
  - **Interviews et ateliers collaboratifs :** Organiser des sessions avec les experts métiers pour comprendre les processus existants, les besoins et les problématiques. Ces discussions doivent être structurées autour de scénarios concrets.
    - *Exemple :* Dans une entreprise de logistique, des ateliers pourraient se concentrer sur le processus d'expédition des colis, en identifiant les étapes critiques comme la gestion des retards ou des erreurs de livraison.
  - **Documentation :** Recueillir et analyser les documents clés (procédures, workflows, rapports) pour compléter les informations obtenues lors des échanges. Cela peut s'illustrer par l'étude des rapports d'incidents ou des manuels de formation pour identifier des points récurrents nécessitant une modélisation claire.
  - **Identification des concepts fondamentaux :** Établir une liste initiale des termes métiers et des concepts critiques à expliciter.
    - *Exemple :* Dans un système bancaire, on pourrait définir des concepts comme le "compte courant", le "solde minimum" ou le "plafond de retrait" comme des éléments essentiels à modéliser.
- **Alignement technique** :

- **Identification des contraintes techniques :** Déterminer les exigences techniques préexistantes, comme les systèmes hérités ou les performances attendues. Une application déjà existante pourrait imposer des contraintes sur les formats de données échangés entre modules ou sur les temps de réponse pour des processus critiques/complexes.
- **Alignement sur les objectifs stratégiques :** Assurer que les objectifs métiers et techniques sont compatibles et qu'ils contribuent à une vision commune.
  - *Exemple :* si l'objectif stratégique d'une plateforme e-commerce est de réduire les retours produits, les objectifs techniques incluraient des fonctionnalités comme l'amélioration des descriptions de produits ou la vérification des tailles avant l'achat pour éviter cela.

2. Élaboration d'un langage commun entre les équipes métier et technique

Une fois les objectifs métiers définis, il est essentiel d'établir un **langage commun** qui sera utilisé par l'ensemble des parties prenantes (experts techniques, experts métiers…) dans leur vocabulaire à la fois dans **les discussions**, la planification des fonctionnalités à développer, mais également dans **le code source**.

Élaboration du glossaire :

L'élaboration de ce langage débute par la création d'un **glossaire**, un document dans lequel les termes métier sont définis avec précision. Par exemple, dans une application e-commerce, le terme **« commande »** pourrait être décrit comme **« un ensemble de produits sélectionnés par un client en vue d'un achat »**. Ces définitions sont validées avec les experts métiers pour garantir leur validité. Le glossaire est ensuite partagé à travers un outil collaboratif comme Notion ou Confluence, afin que toutes les parties puissent y accéder facilement et le mettre à jour selon les évolutions du projet ou du métier lui-même.

Term	Definition
Client	A client is someone, who has ordered at least one item in the past or who has a pending order.
Order	An order is a collection of items that a client has ordered and that are shipped as a package.
Items	A single article with a defined price.

Intégration dans le développement :

Ce langage ubiquitaire ne se limite pas à des mots sur papier. Il doit être intégré dans chaque aspect du développement, notamment dans le code. Les noms des **classes**, des **méthodes** et des **variables** doivent refléter les termes du glossaire, ce qui garantit une cohérence entre les

discussions stratégiques et les implémentations techniques. Par exemple, une classe intitulée Commande pourrait contenir des attributs comme Client ou ListeProduits, reprenant directement les concepts clés du langage ubiquitaire.

Ceci s'applique également à la création des tickets, car la gestion de ce type de projet est indissociable d'une approche **Agile**. Dans ce contexte, les **user stories** doivent intégrer ce langage commun afin de garantir une compréhension partagée par toutes les parties. Par exemple, une user story pourrait être rédigée ainsi : « *En tant que client, je veux pouvoir consulter la liste de mes commandes passées, afin de suivre leur état de livraison*. » Une telle formulation utilise des termes déjà validés dans le glossaire et facilite la transition entre la définition des besoins métiers et leur mise en œuvre technique. Cela simplifie non seulement la définition des besoins, mais aussi leur numérisation par les développeurs, car c'est l'essence même de l'informatique.

Pour préserver cette cohérence, des points de synchronisation réguliers sont organisés entre les équipes métier et technique. Ces réunions permettent de résoudre rapidement les malentendus et d'ajuster les définitions au besoin. Ainsi, le langage ubiquitaire reste évolutif et pertinent tout au long du projet. Ce processus crée un lien direct entre la compréhension métier et le code, évitant les divergences et améliorant la qualité globale du système.

3. Cartographie des bounded contexts et identification des sous-domaines stratégiques

Une fois les informations clés recueillies et un langage commun établi, l'étape suivante consiste à organiser et modéliser le domaine. Cette phase garantit une gestion structurée des différents concepts, souvent complexes.

- **Identification des bounded contexts :**
  - **Analyse des interactions :** il faut identifier les parties du domaine qui interagissent de manière récurrente entre elles et celles qui sont indépendantes. Chaque bounded context représente une unité cohérente de responsabilité.
    - *Exemple :* dans une **plateforme SaaS**, un bounded context pourrait se focaliser sur la gestion des abonnements, tandis qu'un autre se chargerait de la facturation.
  - **Découpage fonctionnel :** L'objectif est de structurer le système de manière logique, en organisant les fonctionnalités en **blocs indépendants** ou **faiblement couplés**, afin de faciliter sa conception, son développement, sa maintenance et son évolution.
    - *Exemple :* dans une plateforme e-commerce, les bounded contexts pourraient inclure "Gestion des commandes", "Catalogue produit" et "Paiements".

Une fois les bounded contexts définis, il est important de classer les sous-domaines selon leur importance stratégique pour l'organisation. Cela aide à prioriser les efforts de développement et à concentrer les ressources sur les parties les plus critiques.

- **Classification des sous-domaines :**
  - **Sous-domaines stratégiques :** les sous-domaines stratégiques sont ceux qui apportent une réelle valeur ajoutée à l'entreprise et qui peuvent constituer un avantage compétitif.
    - *Exemple* : dans une application de covoiturage, un système de suggestion de trajets basé sur un algorithme d'optimisation avancé pourrait être classé comme un sous-domaine stratégique.

- **Sous-domaines de support :** À l'inverse, les sous-domaines de support sont nécessaires, mais ne constituent pas un différenciateur.
  - *Exemple* : dans le même contexte de covoiturage, la gestion des paiements ou des comptes utilisateurs pourrait être considérée comme un sous-domaine de support.

Éléments clés de la modélisation :

La modélisation d'un domaine avec le DDD repose sur des éléments fondamentaux qui permettent de structurer et de gérer la complexité tout en respectant les règles métier.

1. **Les aggregate roots** représentent les entités centrales qui regroupent les données ayant des attributs similaires. Par exemple, dans un système de gestion de bibliothèque, un agrégat « Livre » pourrait inclure des informations sur les emprunts en cours et les futures réservations. Ces agrégats sont le point d'entrée pour toutes les interactions dans leur contexte respectif, garantissant ainsi la cohérence des données.
2. **Les services du domaine** encapsulent les règles métier qui ne sont pas directement liées à une entité spécifique. Par exemple, dans une plateforme de commerce électronique, un service de calcul de réduction pourrait être un service du domaine, utilisé par plusieurs agrégats comme « Commande » ou « Produit ».
3. **Les repositories** agissent comme des interfaces pour accéder et manipuler les données persistantes tout en maintenant une isolation claire entre le domaine et l'infrastructure.
4. **Les spécifications** formalisent les critères de recherche ou de validation, permettant de créer des requêtes complexes tout en restant aligné sur les règles métier.
5. Enfin, les **événements du domaine** capturent les changements significatifs dans le système, comme la confirmation d'une commande ou l'expiration d'un abonnement. Ces événements permettent de communiquer efficacement entre les différents bounded contexts ou microservices.

Cette approche structurée de gestion de projet avec le DDD garantit une meilleure compréhension du domaine, une collaboration efficace entre les équipes métier et technique, ainsi qu'une organisation claire des responsabilités. Ces étapes posent les bases solides pour un développement logiciel aligné sur les besoins métiers et techniquement robuste. Plusieurs éléments de cette section avaient déjà été évoqués précédemment. Cependant, il est essentiel de les redéfinir afin d'avoir une compréhension complète de la phase de planification.

B. Construction de l'équipe

La réussite d'un projet DDD dépend en grande partie de la composition et de la dynamique de l'équipe. Une équipe bien structurée doit comprendre des rôles clés, chacun apportant des compétences spécifiques pour assurer l'alignement entre les objectifs métiers et techniques. Il est également essentiel que chaque membre se sente valorisé dans le projet. En effet, pour des projets à forts enjeux et de grande responsabilité, un manque de visibilité sur la direction du projet peut rapidement affecter le bien-être des acteurs concernés et influencer la qualité du projet.

1. Les divers rôles dans l'équipe

Les **domain experts** jouent un rôle central en fournissant leur expertise métier. Ce sont eux qui définissent les concepts et les règles métiers qui orientent la modélisation du domaine.

Leur collaboration étroite avec les développeurs est essentielle pour maintenir une cohérence entre les besoins métier et les solutions techniques proposées.

Les **développeurs**, quant à eux, traduisent les concepts définis par les domain experts en code. Leur responsabilité ne se limite pas à l'écriture du code : ils participent également à la conception du domaine et doivent comprendre les nuances des règles métiers pour les intégrer correctement.

Les **architectes** ont une vision globale du système. Ils veillent à ce que les choix techniques, comme l'infrastructure ou les frameworks utilisés, soient cohérents avec les principes du DDD. Leur rôle inclut également la définition des bounded contexts et la supervision des interactions entre eux.

En plus de cela, un ou plusieurs acteurs devront occuper le(s) rôle(s) de **Lead Développeur/Expert Technique.** Il sera donc le pilier central dans l'équipe de développement, tant du point de vue technique que stratégique. Ce rôle ne se limite pas seulement à la gestion des aspects de développement, mais s'étend également à l'architecture, la définition des bonnes pratiques et le mentorat des autres membres de l'équipe. Il représente une référence technique pour l'ensemble du groupe, assurant que les choix techniques sont alignés avec les meilleures pratiques et les objectifs de l'entreprise. En termes de compétences, il est crucial qu'il possède non seulement une expertise approfondie en DDD, mais aussi une maîtrise de **l'architecture logicielle** et des **design patterns**. Ces compétences techniques sont essentielles pour créer des solutions robustes, évolutives et bien structurées.

Cependant, ces compétences techniques ne suffisent pas à garantir la réussite d'un projet. Les **soft skills** jouent un rôle tout aussi important, en particulier dans des projets DDD complexes. Il devra donc posséder des compétences interpersonnelles telles que le **mentorat** qui est essentiel pour accompagner les membres moins expérimentés, leur transmettre des connaissances et créer un environnement de travail collaboratif. Cela contribue non seulement au développement des compétences individuelles, mais aussi à la cohésion de l'équipe et à la résolution collective de problèmes.

Le DDD (Domain-Driven Design), étant un principe relativement récent, peut parfois être perçu comme difficile à maîtriser pour un grand nombre de personnes. Il peut susciter de la crainte, voire de la résistance, chez certains experts du domaine informatique. En effet, bien qu'ils soient attentifs aux nouvelles approches, ces derniers restent souvent attachés à leurs anciennes pratiques, renforcées par des habitudes bien ancrées. C'est pourquoi il est essentiel de faire preuve d'une pédagogie adaptée pour les aider à comprendre et à adhérer au DDD.

2. La méthodologie de gestion de projet

Pour une collaboration optimale, il est crucial d'adopter des méthodologies adaptées comme **Scrum** ou **Kanban**. Selon nous, la méthodologie Scrum est la plus optimale pour notre cas. Avec ses sprints et ses réunions structurées, il permet de maintenir une cadence régulière tout en favorisant les ajustements rapides en fonction des besoins métiers changeants. Les **daily** facilitent la communication entre les membres de l'équipe, tandis que les rétrospectives encouragent une amélioration continue. Pour ce faire, il sera nécessaire d'avoir un **Product Owner** du côté des experts métier, ainsi qu'un **Scrum Master** qui jouera le rôle de régisseur, coordonnant et facilitant le processus.

C. Gestion des autres phases du projet et études de cas : succès et échecs

La phase d'analyse ayant déjà été abordée précédemment en détail, nous ne reviendrons pas dessus. Nous aborderons donc la phase de conception, qui consiste à choisir les outils et les technologies. Nous discuterons de son importance dans le projet, bien que le DDD offre une certaine flexibilité à ce niveau. Enfin, nous parlerons des phases de développement et d'analyse continue, au cours desquelles nous faisons le point sur les travaux réalisés et à venir, afin d'assurer la maintenance et l'évolution de la solution.

1. Phase de conception : définition des APIs, choix technologiques et outils

La phase de conception marque le pont entre l'abstraction des idées et la concrétisation technique. C'est ici que l'imagination prend forme, que les choix stratégiques sont faits et que chaque décision devient une brique essentielle pour construire un système robuste et évolutif. Ce moment crucial requiert autant de rigueur que de créativité.

Définition des APIs : Contract-first

Créer une API, ce n'est pas seulement exposer des données ou des fonctionnalités. C'est bâtir une interface qui deviendra le langage commun entre les parties prenantes : développeurs, systèmes et utilisateurs finaux. Pour garantir cette fluidité, nous adoptons une approche contractuelle solide :

- **Contract-first** : Avant même d'écrire une seule ligne de code, nous définissons les contrats d'API, souvent à l'aide de standards comme **OpenAPI** ou **GraphQL**. Cela établit une base claire et partagée sur laquelle tout le développement reposera.
- **Prise en compte des bounded contexts** : dans l'esprit du Domain-Driven Design, chaque API se limite à un contexte bien défini, assurant une séparation nette des responsabilités. Cette stratégie empêche les dépendances excessives et préserve l'agilité du système à long terme.

Choix technologiques :

Le choix des technologies n'est pas un exercice arbitraire, mais une réponse réfléchie aux besoins fonctionnels et non fonctionnels du domaine. Dans notre cas, voici les piliers techniques qui soutiendront notre architecture :

- **TypeScript** : un langage fort typé qui n'est pas seulement une assurance contre les erreurs, mais aussi une documentation vivante du code. Il garantit une collaboration fluide au sein de l'équipe, surtout dans un environnement complexe comme celui du DDD.
- **NodeJS** : sa modularité et son vaste écosystème en font un allié précieux pour construire des systèmes scalables et performants, tout en respectant les principes DDD.

Et bien sûr, ces technologies s'accompagnent d'outils complémentaires indispensables :

- **Bases de données adaptées** : nous privilégierons PostgreSQL pour des structures relationnelles robustes ou MongoDB pour des besoins NoSQL flexibles, selon les contraintes du domaine.
- **Outils de CI/CD** : GitLab CI ou Jenkins pour automatiser, tester et déployer avec une rapidité et une fiabilité maximales.

Résultats attendus :

Au terme de cette phase, nous n'aurons pas seulement un ensemble d'outils et d'interfaces. Nous aurons une architecture claire, qui incarne les besoins du domaine et s'aligne harmonieusement avec les principes du Domain-Driven Design. Chaque choix technologique sera documenté, justifié et intégré comme une réponse précise aux contraintes identifiées, qu'elles soient fonctionnelles ou non fonctionnelles.

2. Phase de développement : application des pratiques DDD, gestion des changements et respect des contraintes métier

C'est ici que la vision conceptualisée prend une forme tangible. La phase de développement est bien plus qu'un simple processus de codage, c'est un alliage entre les exigences métier, les contraintes techniques et les principes du Domain-Driven Design. Chaque ligne de code devient une brique essentielle qui respecte le modèle conçu, tout en s'adaptant aux changements inévitables du projet.

Application des pratiques DDD : structurer l'essence du métier

Le DDD est le cœur de cette étape. Il nous guide pour bâtir un logiciel qui parle le langage du domaine, tout en restant robuste et maintenable.

- **Les tactiques en action** :
    - **Implémentation des agrégats et des événements de domaine** : chaque agrégat devient un bloc autonome qui encapsule les règles métier, tandis que les événements de domaine permettent de modéliser les changements significatifs dans le système.
    - **Utilisation des repositories** : ces composants gèrent la persistance tout en préservant l'intégrité du domaine.
- **Les méthodes incontournables** :
    - **Tests unitaires et end-to-end (E2E) :** Les tests ne sont pas seulement un filet de sécurité, mais un véritable levier pour valider les comportements et garantir la qualité du code. Il est possible de les automatiser à l'aide d'outils comme **Cypress,** qui sera inclus dans la composante qui s'occupe de l'intégration continue.
    - **Refactorisation continue** : parce qu'un modèle conceptuel n'est jamais figé, le code évolue constamment pour rester fidèle aux besoins du domaine, tout en éliminant les dettes techniques.

Gestion des changements :

Dans tout projet, les changements sont inévitables. Ce n'est pas une menace ou un danger, mais une opportunité d'amélioration. L'objectif est d'intégrer ces évolutions tout en maintenant la stabilité et la cohérence du système.

- **Adaptabilité** : les retours métier, qu'ils viennent des utilisateurs finaux ou des experts du domaine, sont intégrés de manière fluide pour enrichir l'application sans introduire de chaos.
- **Outils clés** : le versionnage des APIs est une pratique indispensable pour gérer les modifications tout en évitant de perturber les consommateurs existants.

Respect des contraintes métier :

Sachant que les règles métier sont l'âme de l'application. Leur respect est inéluctable, même face à des évolutions techniques ou des ajustements d'architecture.

- **Mise en place de garde-fous** : Ces mécanismes assurent que chaque modification reste alignée avec les objectifs stratégiques et les règles fondamentales du domaine. Qu'il s'agisse de validations, de contrôles ou de processus automatisés, tout est pensé pour préserver l'intégrité du système.

La phase de développement n'est pas qu'une simple exécution technique. C'est un moment où chaque choix reflète une compréhension profonde du domaine, où chaque ligne de code contribue à la fois à résoudre des problèmes immédiats et à poser les bases d'une solution durable. L'objectif ultime étant de livrer un logiciel fonctionnel, maintenable et fidèle aux principes du DDD. Cette phase est bien entendu aussi portée par les bienfaits de la méthodologie Agile Scrum.

3. Analyse d'un projet bien mené avec le DDD : HR Management System

Dans ce cas de figure où nous allons analyser les raisons du succès d'un projet mené avec le DDD, prenons l'exemple d'une application de gestion des ressources humaines que l'on va appeler HRMS pour ***"HR Management System"***.

Ce projet est un modèle de réussite, car il a su intégrer le Domain-Driven Design (DDD) de manière fluide et structurée. Il s'agit d'un projet où chaque décision technique a été pensée pour répondre aux besoins du métier tout en permettant une évolutivité future.

Les succès clés :

1. **Un modèle clair et un langage commun (Ubiquitous Language)** : Le projet a débuté par l'élaboration d'un modèle clair qui a permis à l'ensemble de l'équipe de développeurs, de métiers et de parties prenantes de parler le même langage. Chaque terme utilisé dans l'application avait une signification précise et partagée, créant ainsi une cohérence parfaite entre les équipes techniques et fonctionnelles.
2. **Réduction significative des bugs grâce à une architecture bien structurée** : En appliquant les principes du DDD, le système a été conçu de manière à minimiser les dépendances et à renforcer l'intégrité des données. Cela a permis d'identifier et de résoudre les erreurs rapidement, avec un impact minimal sur l'application globale.
3. **Des résultats tangibles : Livrables dans les temps, système évolutif et maintenable** : Non seulement les livrables ont été fournis dans les délais, mais l'architecture adoptée a permis une grande flexibilité, rendant l'application facilement extensible à mesure que les besoins du métier évoluent.

Cette réussite repose sur la capacité à lier les aspects techniques aux objectifs métier, ce qui a permis de créer un produit de qualité, aligné avec les besoins des utilisateurs finaux tout en respectant les contraintes techniques.

4. Exemples de défis rencontrés et solutions proposées

Il n'est pas sans savoir que le chemin de tout projet n'est jamais linéaire, et les projets intégrant les principes du DDD ne font pas exception. Voici quelques défis majeurs très souvent rencontrés et les solutions adoptées pour les surmonter :

Scope creep : gérer les attentes et les changements

- **Défi** : le scope creep est l'extension progressive et non maîtrisée du périmètre initial. Cela survient souvent lorsque de nouvelles fonctionnalités sont ajoutées sans évaluation approfondie de leur impact sur le projet.

- **Solution** : la clé est de mettre en place des **contrats clairs** avec les parties prenantes dès le début. Ces contrats définissent des attentes précises et permettent de maintenir un backlog bien hiérarchisé. Cela permet de garder une vue d'ensemble et de hiérarchiser les demandes, assurant que seules les évolutions les plus critiques seront priorisées.

Mauvaise définition des bounded contexts : clarifier les responsabilités

- **Défi** : un autre obstacle est le chevauchement des responsabilités entre les différents **bounded contexts**. Lorsque les délimitations des contextes ne sont pas claires, cela peut entraîner des incohérences et des conflits dans le système.
- **Solution** : le travail avec les **experts métier** est donc crucial ici. Une révision minutieuse des délimitations des contextes va permettre de bien définir les frontières et d'éviter les interdépendances inutiles. Des **refactorisations ciblées** sont à effectuer pour garantir que chaque partie du système reste autonome et clairement définie.

Problèmes de communication : créer une compréhension partagée

- **Défi** : la mauvaise interprétation des exigences métier est un défi fréquent dans les projets complexes, où chaque partie prenante peut avoir une vision différente du problème.
- **Solution** : pour résoudre ce problème, des **ateliers réguliers** sont organisés. Ces sessions permettent de s'assurer que tous les membres de l'équipe ont une **compréhension commune** des exigences et des objectifs. Ces moments de partage renforcent la collaboration entre les équipes techniques et métiers, assurant ainsi un meilleur alignement au fur et à mesure de l'avancement du projet.

Ces défis ne sont pas des obstacles insurmontables, mais des occasions d'apprentissage. Chaque problème rencontré a conduit à une amélioration continue des pratiques de développement, renforçant ainsi la capacité du projet à s'adapter aux réalités du métier tout en préservant sa cohérence.

Conclusion :

A. Synthèse

En conclusion, notre cours intitulé "TypeScript et Domain-Driven Design pour le développement d'API sur Node.js" a permis de combiner deux approches essentielles et complémentaires pour répondre aux exigences croissantes en matière de robustesse, de maintenabilité et d'évolutivité des systèmes logiciels modernes.

À travers une exploration approfondie, nous avons démontré que TypeScript offre des avantages significatifs en termes de sécurité et de structuration du code. Son typage statique robuste, ses fonctionnalités avancées (comme les interfaces, les classes et les generics) et sa capacité à prévenir les erreurs à la compilation en font un outil indispensable pour les projets complexes et collaboratifs. Ces qualités facilitent également la maintenance des applications à grande échelle, tout en garantissant une meilleure lisibilité et collaboration entre les développeurs.

En parallèle, l'intégration du Domain-Driven Design a mis en lumière l'importance de placer le domaine métier au cœur du processus de développement. Grâce à des concepts clés comme le langage ubiquitaire, les contextes bornés et les agrégats, nous avons démontré

comment le DDD permet de modéliser des systèmes complexes, tout en assurant une cohérence et une adaptabilité face aux changements. Cette méthodologie, en créant un pont solide entre les besoins métiers et les solutions techniques, s'est avérée être un atout majeur pour concevoir des architectures modulaires et résilientes.

La mise en pratique de ces concepts a été réalisée à travers une démonstration d'un système de gestion d'API, structuré autour des couches Domaine, Application et Infrastructure. Le choix de technologies comme Node.js, Express.js et TypeORM a permis une intégration fluide et pragmatique des principes du DDD, tout en restant accessible et adapté à des projets nécessitant légèreté et modularité. Bien que d'autres frameworks comme NestJS puissent offrir des fonctionnalités plus intégrées, notre approche a montré qu'un stack minimaliste pouvait tout autant répondre aux exigences du DDD.

Ce projet nous a permis d'acquérir des compétences théoriques et pratiques essentielles pour le développement d'API modernes. En particulier, nous avons appris à :

- **Allier théorie et pratique**, en appliquant des concepts abstraits dans un cadre concret.
- **Collaborer efficacement**, en établissant une communication claire entre les développeurs et les experts métiers.
- **Adopter une approche modulaire**, assurant évolutivité et résilience face aux défis futurs.

Ces apprentissages constituent un socle solide pour nos projets futurs, que ce soit dans un cadre académique ou professionnel. En outre, les outils et méthodologies explorés dans ce projet offrent des perspectives riches pour adresser les besoins complexes des systèmes modernes, où la robustesse et l'adaptabilité sont devenues des impératifs.

En somme, **TypeScript et le Domain-Driven Design**, loin d'être simplement des outils ou méthodologies, se présentent comme des leviers stratégiques pour concevoir des solutions logicielles pérennes, alignées sur les attentes métier et résilientes face aux évolutions technologiques. Ce projet a non seulement renforcé notre expertise, mais il a également posé les bases d'une démarche professionnelle centrée sur l'excellence et la collaboration.

Réflexions et recherches

Bibliographie

Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley, 2003.
Une référence incontournable qui expose les fondements théoriques et pratiques du DDD, centrée sur l'approche métier et les modèles stratégiques.

Vaughn Vernon, *Implementing Domain-Driven Design*, Addison-Wesley, 2013.
Complète l'approche DDD en fournissant des exemples pratiques et des implémentations spécifiques.

Khalil Stemmler, *Advanced TypeScript: Volumes 1 & 2*.
Guide avancé sur TypeScript et son application avec DDD.

Nathan Rozentals, *Mastering TypeScript*, Packt Publishing, 2021.
Un manuel détaillé sur TypeScript, couvrant à la fois les bases et les concepts avancés.

Webographie

TypeScript

Site officiel de TypeScript
Documentation complète et tutoriels pour apprendre et utiliser TypeScript.

Node.js avec TypeScript
Introduction officielle pour démarrer un projet Node.js avec TypeScript.

YouTube - Introduction à TypeScript
Tutoriel vidéo pour débutants.

YouTube - TypeScript avancé
Exploration des fonctionnalités avancées.

W3Schools - Apprendre TypeScript

Tutoriel interactif pour apprendre les bases..

Node.js

API Node.js - Documentation
Documentation officielle pour les développeurs Node.js.

W3Schools - Introduction à Node.js
Tutoriels pratiques sur les bases de Node.js.

OpenClassrooms - Full Stack avec Node.js
Un cours complet pour construire des applications Node.js.

Domain-Driven Design

Les Dieux du Code - Introduction au DDD
Un article clair pour comprendre les bases du Domain-Driven Design.

Alex so yes - DDD (Théorie et Pratique)
Exploration approfondie du DDD, avec un focus sur les patterns stratégiques et tactiques.

Khalil Stemmler - Blog sur DDD et TypeScript
Cas pratiques et exemples sur l'application du DDD avec TypeScript.

Microsoft Learn - Microservices et DDD
L'intégration du DDD dans des architectures modernes.

GitHub - Exemple de projet Node.js avec DDD
Un dépôt github d'exemple pour comprendre l'intégration du DDD avec Node.js.

Livre électronique : *Domain-Driven Design Quickly*
Résumé concis des principes du livre d'Eric Evans.

Medium - DDD avec Node.js
Application pratique du DDD avec Node.js.

Glossaire :

TypeScript

- **TypeScript** : Langage de programmation open-source, sur-ensemble typé de JavaScript, qui améliore la robustesse et la maintenabilité du code.
- **Typage statique** : Définition explicite des types des variables et des fonctions, permettant de détecter les erreurs à la compilation.
- **Interface** : Structure décrivant la forme des objets, incluant leurs propriétés et types.
- **Classe** : Modèle utilisé pour créer des objets, encapsulant des propriétés et des méthodes.
- **Transpilation** : Processus de conversion du code TypeScript en JavaScript pour assurer la compatibilité avec les navigateurs.
- **Decorators** : Fonctions spéciales ajoutant des métadonnées ou modifiant le comportement des classes, méthodes ou propriétés.
- **Generic Types** : Types réutilisables qui fonctionnent avec plusieurs types de données.
- **Namespace** : Espace de nommage utilisé pour organiser le code et éviter les conflits de noms dans de grands projets.

Node.js

- **Node.js** : Environnement d'exécution JavaScript côté serveur, conçu pour construire des applications web performantes et évolutives.
- **API (Application Programming Interface)** : Interface permettant l'interaction entre différents logiciels ou services.
- **Asynchrone** : Mode de programmation non bloquant, permettant de gérer plusieurs tâches en parallèle.
- **Event-Driven** : Modèle d'architecture basé sur des événements, où les actions sont déclenchées en réponse à des événements.
- **NPM (Node Package Manager)** : Gestionnaire de paquets de Node.js pour installer, partager et gérer les bibliothèques JavaScript.
- **Middleware** : Logiciel intermédiaire dans Express.js qui traite les requêtes avant d'atteindre les routes définies.
- **Cluster** : Module natif de Node.js permettant d'exploiter plusieurs cœurs d'un processeur pour améliorer les performances.
- **Stream** : Interface pour lire ou écrire des données en continu (par exemple, fichiers ou requêtes HTTP).
-

Express.js

- **Express.js** : Framework web minimaliste et flexible pour Node.js, utilisé pour créer des applications et des APIs.
- **Route** : Définition d'un chemin d'accès spécifique dans une API pour répondre à une requête donnée.
- **Middleware** : Fonction intermédiaire exécutée avant la logique finale d'une route, utilisée pour des tâches comme l'authentification ou la gestion des erreurs.

Domain-Driven Design (DDD)

- **DDD (Domain-Driven Design)** : Méthodologie de conception axée sur la modélisation du domaine métier pour aligner les solutions techniques sur les besoins réels.
- **Bounded Context** : Partie isolée d'un domaine avec des règles métier propres, évitant les conflits entre différents sous-domaines.
- **Langage ubiquitaire (Ubiquitous Language)** : Vocabulaire commun partagé entre développeurs et experts métier pour assurer une compréhension mutuelle et éviter les ambiguïtés.
- **Entité** : Objet avec une identité unique, qui reste constante au fil du temps et des modifications.
- **Valeur Objet (Value Object)** : Objet défini uniquement par ses attributs, sans identité propre, généralement immuable.
- **Agrégat (Aggregate)** : Regroupement logique d'entités et d'objets de valeur formant une unité cohérente avec une racine d'agrégat (Aggregate Root).
- **Service de Domaine** : Composant encapsulant une logique métier complexe qui ne peut pas être associée à une seule entité ou agrégat.
- **Événements de Domaine** : Notifications représentant des changements significatifs dans le domaine, utilisées pour communiquer entre les contextes.
- **Repository** : Interface pour accéder aux données persistantes tout en préservant l'isolation du domaine.
- **Specification Pattern** : Design pattern permettant de formaliser les critères de recherche ou de validation dans le domaine.

Autres concepts importants

- **ORM (Object Relational Mapper)** : Outil permettant de manipuler les bases de données en utilisant des objets dans le code (ex. : TypeORM).
- **CQRS (Command Query Responsibility Segregation)** : Pattern séparant les responsabilités de lecture et d'écriture dans une application pour améliorer les performances et la maintenabilité.
- **Event Sourcing** : Modèle où chaque changement d'état est enregistré sous forme d'événements immuables pour assurer une traçabilité complète.
- **Swagger** : Outil permettant de documenter les APIs de manière interactive et normalisée.
- **Docker** : Plateforme de conteneurisation utilisée pour déployer des applications de manière isolée et portable.
- **Kubernetes** : Outil d'orchestration de conteneurs pour gérer le déploiement, la mise à l'échelle et la maintenance des applications conteneurisées.