CESI
ÉCOLE D'INGÉNIEURS

Prosit 4 : Des données à exploiter

Ritter Antoine

Bloc Elective Développement web avancé

10 juin 2025

CESI École d'Ingénieur
Campus de Nancy
Promotion FISE Informatique 23-26

Table des matières

Chapitre 1

Prosit Aller

1.1 Prendre connaissance de la situation et la clarifier

1.1.1 Contexte

Notre but est de développer une interface utilisateur exploitant des données hétérogènes de capteurs nécessitant un traitement en JavaScript pour garantir une cohérence dans une base de données NoSQL.

1.1.2 Mots-clés

— Base de données (BDD)
— Multiparadigme
— Format clé-valeur
— Base NoSQL
— BDD orientée document
— SSH
— Backend
— Requête classique
— Flux de données en temps réel
— Surcouche
— Interface de gestion de BDD
— JavaScript
— Moteur V8
— Relation applicative
— Serveur central

1.2 Analyse du besoin

1.2.1 Problème(s)

— Comment traiter des données hétérogènes ?

1.2.2 Contraintes

— Format des données
— Connexion en SSH
— Format des transmissions

1.2.3 Livrables

— Code en JavaScript pour le backend
— Frontend
— Une base de données NoSQL
— Simulation des capteurs

1.3 Généralisation du problème

— Formatage de données
— Communication back-front

1.4 Pistes de solutions

— Gérer le décalage horaire
— Gérer les formats en amont de la base de données
— On ne peut pas automatiser l'ajout des capteurs à 100%
— Unifier le format des capteurs est la manière solution pour éradiquer les données inexploitables
— Le NoSQL permet de stocker plus de formats de données par rapport au SQL

1.5 Plan d'action

1. Se renseigner sur les bases de données NoSQL
2. Se renseigner sur les formats de données
3. Protocoles de transmission
4. Technologies types : Express, Node.js, TypeScript
5. Triangle de CAP
6. 5V du Big Data

Chapitre 2

Notions de cours

2.1 Mots Clés

2.1.1 Base de données (BDD)

Une base de données est un ensemble structuré de données stockées et organisées de manière à être facilement accessibles et gérables.

2.1.2 Multiparadigme

Le multiparadigme est un langage de programmation ou une approche qui prend en charge plus d'un paradigme de programmation, comme la programmation orientée objet, fonctionnelle, etc.

2.1.3 Format clé-valeur

Le format clé-valeur est un modèle de données où chaque élément est stocké comme une paire attribut-valeur, où la clé est un identifiant unique.

2.1.4 Base NoSQL

Une base NoSQL est une base de données qui fournit un mécanisme pour le stockage et la récupération de données qui est différent des tables relationnelles utilisées dans les bases de données SQL.

2.1.5 BDD orientée document

Une BDD orientée document est une base de données qui stocke des informations sous forme de documents, généralement au format JSON ou XML, plutôt que dans des tables relationnelles.

2.1.6 SSH (Secure Shell)

SSH est un protocole de réseau cryptographique pour l'exploitation sécurisée de services réseau sur un réseau non sécurisé.

2.1.7 Backend

Le backend est la partie d'un système ou d'une application qui s'occupe de la logique métier, de l'accès aux données, de l'authentification des utilisateurs, etc., et qui n'est pas directement accessible par l'utilisateur final.

2.1.8 Requête classique

Une requête classique est une demande d'information ou d'action envoyée à une base de données ou un serveur, généralement en utilisant le langage SQL pour les bases de données relationnelles.

2.1.9 Flux de données en temps réel

Un flux de données en temps réel est un flux continu de données qui sont traitées et analysées en temps réel, souvent utilisé dans les systèmes de surveillance et d'analyse.

2.1.10 Surcouche

Une surcouche est une couche logicielle ajoutée au-dessus d'une autre pour fournir une interface ou des fonctionnalités supplémentaires.

2.1.11 Interface de gestion de BDD

Une interface de gestion de BDD est un outil ou une application qui permet aux utilisateurs d'interagir avec une base de données pour effectuer des tâches de gestion et de maintenance.

2.1.12 JavaScript

JavaScript est un langage de programmation de haut niveau, principalement utilisé pour rendre les pages web interactives.

2.1.13 Moteur V8

Le moteur V8 est un moteur JavaScript open source développé par Google, utilisé dans les navigateurs Chrome et Node.js pour exécuter du code JavaScript.

2.1.14 Relation applicative

La relation applicative est la manière dont différentes applications ou composants logiciels interagissent et communiquent entre eux.

2.1.15 Serveur central

Un serveur central est un serveur qui agit comme un point central pour le stockage des données, la gestion des ressources, et la coordination des activités dans un réseau.

2.2 Introduction aux Bases de Données NoSQL

Les bases de données NoSQL (Not Only SQL) sont conçues pour répondre aux besoins des applications modernes qui nécessitent une grande flexibilité, une évolutivité horizontale et des performances élevées. Contrairement aux bases de données relationnelles, les bases NoSQL ne suivent pas le schéma tabulaire traditionnel et sont particulièrement adaptées pour gérer de grands volumes de données non structurées ou semi-structurées.

2.2.1 Types de Bases de Données NoSQL

- **Bases de données orientées documents** : Stockent les données sous forme de documents, généralement au format JSON. Exemples : MongoDB, CouchDB.
- **Bases de données clés-valeurs** : Utilisent un modèle simple où chaque élément est stocké comme une paire clé-valeur. Exemples : Redis, DynamoDB.
- **Bases de données colonnes larges** : Stockent les données dans des colonnes plutôt que dans des lignes, optimisant ainsi les requêtes d'agrégation. Exemples : Cassandra, HBase.
- **Bases de données orientées graphes** : Utilisent des structures de graphes pour stocker les données, idéales pour les relations complexes. Exemples : Neo4j, ArangoDB.

2.2.2 Cas d'Utilisation

Les bases de données NoSQL sont idéales pour les applications nécessitant une grande flexibilité dans le schéma de données, une évolutivité horizontale facile, et des performances élevées en lecture/écriture. Elles sont souvent utilisées dans les applications web modernes, les systèmes de gestion de contenu, et les applications IoT.

2.3 Comprendre les Formats de Données

Les formats de données jouent un rôle crucial dans la manière dont les informations sont stockées et échangées entre systèmes. Voici une analyse détaillée des formats les plus courants :

2.3.1 JSON (JavaScript Object Notation)

JSON est un format de données léger et facile à lire/écrire pour les humains, tout en étant facile à analyser et générer pour les machines. Il est largement utilisé pour les échanges de données sur le web en raison de sa simplicité et de sa compatibilité avec JavaScript.

```
{
  "nom": "Antoine",
  "age": 21,
  "ville": "Nancy"
}
```

Listing 2.1 – Exemple de JSON

2.3.2 XML (eXtensible Markup Language)

XML est un langage de balisage qui définit un ensemble de règles pour encoder des documents dans un format qui est à la fois lisible par l'homme et par la machine. Il est utilisé pour sa capacité à définir des structures de données complexes et pour son extensibilité.

```
<personne>
  <nom>Antoine</nom>
  <age>21</age>
  <ville>Nancy</ville>
</personne>
```

Listing 2.2 – Exemple de XML

2.3.3 CSV (Comma-Separated Values)

CSV est un format de fichier ouvert simple utilisé pour stocker des données tabulaires, où chaque ligne représente un enregistrement et chaque colonne représente un champ.

```
nom,age,ville
Antoine,21,Nancy
```

Listing 2.3 – Exemple de CSV

2.4 Protocoles de Transmission des Données

Les protocoles de transmission sont essentiels pour la communication entre différents systèmes et applications. Ils définissent les règles et les formats utilisés pour l'échange de messages.

2.4.1 HTTP/HTTPS

HTTP (HyperText Transfer Protocol) est un protocole de la couche application utilisé pour la transmission de documents hypertexte, comme HTML. HTTPS est la version sécurisée de HTTP, utilisant le chiffrement SSL/TLS pour sécuriser les communications.

2.4.2 FTP (File Transfer Protocol)

FTP est un protocole de réseau standard utilisé pour le transfert de fichiers d'un hôte à un autre sur un réseau TCP, comme Internet. Il est souvent utilisé pour télécharger des fichiers sur un serveur web.

2.4.3 WebSocket

WebSocket est un protocole de communication qui fournit des canaux de communication full-duplex sur une seule connexion TCP. Il est particulièrement utile pour les applications nécessitant des communications en temps réel, comme les chats en ligne et les jeux multijoueurs.

2.5 Technologies Clés : Express, Node.js, et TypeScript

2.5.1 Express

Express est un framework minimaliste pour Node.js, utilisé pour construire des applications web et des API. Il simplifie le processus de développement en fournissant une série de fonctionnalités robustes pour la gestion des requêtes HTTP, des routes, et des middlewares.

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
  res.send('Hello World!');
});

app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

Listing 2.4 – Exemple de code Express

2.5.2 Node.js

Node.js est un environnement d'exécution JavaScript côté serveur, permettant aux développeurs d'utiliser JavaScript pour le développement backend. Il est particulièrement apprécié pour sa capacité à gérer des opérations d'entrée/sortie de manière asynchrone, ce qui le rend très performant pour les applications en temps réel.

2.5.3 TypeScript

TypeScript est un sur-ensemble typé de JavaScript qui compile en JavaScript simple. Il ajoute des fonctionnalités comme les types statiques, les interfaces, et les classes, ce qui aide à détecter les erreurs plus tôt dans le processus de développement et améliore la maintenabilité du code.

```
interface User {
  name: string;
  age: number;
}

const user: User = {
  name: 'Antoine',
  age: 21
};
```

Listing 2.5 – Exemple de code TypeScript

2.6 Le Triangle de CAP

Le triangle de CAP est un concept fondamental dans la théorie des systèmes distribués. Il stipule qu'il est impossible pour un système distribué de garantir simultanément les trois propriétés suivantes :

— **Cohérence (Consistency)** : Tous les nœuds voient les mêmes données au même moment.

— **Disponibilité (Availability)** : Chaque requête reçoit une réponse non erronée, sans garantie qu'elle contienne les données les plus récentes.
— **Tolérance au partitionnement (Partition tolerance)** : Le système continue à fonctionner malgré des défaillances de communication entre les nœuds.

Les systèmes distribués doivent donc faire des compromis en fonction de leurs besoins spécifiques. Par exemple, un système peut choisir de privilégier la disponibilité et la tolérance au partitionnement au détriment de la cohérence.

2.7 Les 5V du Big Data

Le concept des 5V du Big Data décrit les caractéristiques principales des données massives :
— **Volume** : La quantité massive de données générées chaque seconde.
— **Vélocité** : La vitesse à laquelle ces données sont produites et traitées.
— **Variété** : Les différents types de données disponibles, allant des données structurées aux données non structurées.
— **Véracité** : La qualité et la fiabilité des données.
— **Valeur** : Le potentiel de création de valeur à partir de l'analyse de ces données.

Chapitre 3

Solution

Le code de la solution est disponible sur mon github : Repository.

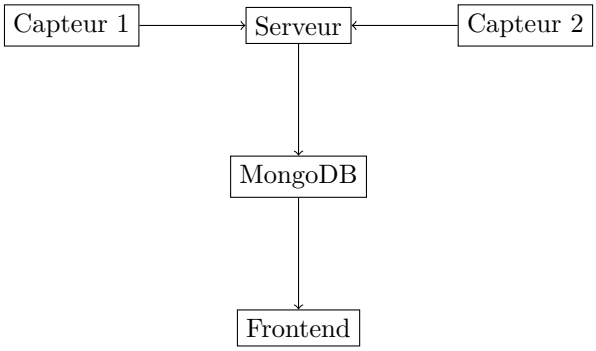
3.1 Structure du Projet

Pour ce projet, j'ai opté pour une architecture basée sur des services Docker. C'est une approche que j'ai trouvée plutôt clean et efficace pour gérer différents composants de manière isolée. Voici les principaux services que j'ai mis en place :

- **mongodb_sensors** : J'ai utilisé une base de données MongoDB pour stocker les données des capteurs. C'est une solution NoSQL qui offre une grande flexibilité pour gérer des données non structurées.
- **sensor1** : Premier capteur qui envoie des données au serveur. J'ai implémenté ce capteur en Python, un langage que je trouve bien adapté pour des applications IoT.
- **sensor2** : Deuxième capteur, similaire au premier, mais avec un identifiant unique différent. Il communique également avec le serveur via des requêtes HTTP.
- **server** : Serveur backend qui reçoit et traite les données des capteurs. J'ai utilisé Node.js avec le framework Express.js pour construire une API RESTful.
- **frontend** : Interface utilisateur pour visualiser les données. J'ai choisi React pour sa réactivité et sa capacité à offrir une expérience utilisateur fluide.

3.2 Diagramme de la Structure

Voici un diagramme qui illustre la structure du projet :



3.3 Fonctionnement et Choix Technologiques

3.3.1 MongoDB

Le service `mongodb_sensors` utilise l'image officielle de MongoDB. J'ai choisi MongoDB pour sa flexibilité et sa capacité à gérer des données non structurées, ce qui est essentiel pour un projet IoT.

— **Ports** : J'ai exposé le port 27017 pour permettre la communication avec la base de données.
— **Volumes** : J'ai utilisé un volume Docker pour persister les données, assurant ainsi que les données ne sont pas perdues lorsque le conteneur est arrêté ou redémarré.

3.3.2 Capteurs

Les services `sensor1` et `sensor2` sont construits à partir du même Dockerfile situé dans le répertoire `./sensor`. Chaque capteur a un identifiant unique et envoie des données au serveur via l'URL spécifiée dans la variable d'environnement `SERVER_URL`.

— **Technologie** : J'ai utilisé Python pour implémenter les capteurs, un langage que je trouve léger et bien adapté pour des applications IoT.
— **Communication** : Les capteurs communiquent avec le serveur via des requêtes HTTP, une méthode simple et efficace pour envoyer des données.

3.3.3 Serveur

Le service `server` est construit à partir du Dockerfile situé dans le répertoire `./backend`. Il expose le port 3000 et dépend du service `mongodb_sensors`. J'ai utilisé la variable d'environnement `MONGO_URI` pour se connecter à la base de données MongoDB.

— **Technologie** : J'ai choisi Node.js avec le framework Express.js pour construire le serveur. C'est une combinaison légère et facile à utiliser pour construire des API RESTful.
— **Base de données** : Le serveur utilise MongoDB pour stocker les données des capteurs, ce qui permet une intégration facile et une gestion efficace des données non structurées.

3.3.4 Frontend

Le service `frontend` est construit à partir du Dockerfile situé dans le répertoire `./frontend`. Il expose le port 3001 et dépend du service `server`.

— **Technologie** : J'ai utilisé React pour implémenter le frontend. C'est un framework moderne qui offre une grande réactivité et une expérience utilisateur fluide.
— **Communication** : Le frontend communique avec le serveur via des requêtes HTTP pour récupérer et afficher les données des capteurs.

3.4 Justification des Choix Technologiques

3.4.1 Docker et Docker Compose

J'ai choisi d'utiliser Docker pour conteneuriser chaque service. Cela permet une isolation des environnements et une gestion simplifiée des dépendances. Docker Compose est utilisé pour orchestrer les différents services, ce qui facilite le déploiement et la gestion de l'application.

— **Isolation** : Chaque service est isolé dans son propre conteneur, ce qui permet de gérer les dépendances et les configurations de manière indépendante.
— **Portabilité** : Les conteneurs Docker peuvent être déployés sur n'importe quelle machine équipée de Docker, ce qui facilite le déploiement et la mise à l'échelle de l'application.

3.4.2 MongoDB

J'ai choisi MongoDB pour sa flexibilité et sa capacité à gérer des données non structurées. C'est une base de données NoSQL qui offre une grande évolutivité et une haute disponibilité, ce qui est essentiel pour les applications IoT.

3.4.3 Frameworks Backend et Frontend

J'ai choisi des frameworks comme Express.js pour le backend et React pour le frontend pour leur légèreté et leur facilité d'utilisation. Ils permettent de construire rapidement des applications robustes et évolutives.