Livrables Backend - Documentation Technique

Sommaire

---

1. Description textuelle des fonctionnalités backend

1.1. Système de filtrage multi-critères

Le système de filtrage permet de rechercher des véhicules selon plusieurs critères qui peuvent être combinés entre eux.

Champs textuels (recherche par correspondance partielle) :

- Marque, Modèle, Couleur, Carburant, Boîte de vitesses, État
- Utilisation de regex MongoDB avec option `i` (insensible à la casse)
- Exemple : `marque: "Peugeot"` → recherche toutes les marques contenant "Peugeot"

Champs numériques avec plages (min/max) :

- Année : `anneeMin` et `anneeMax` → utilisation de `$gte` et `$lte`
- Prix : `prixMin` et `prixMax` → utilisation de `$gte` et `$lte`
- Kilométrage : `kilometrageMin` et `kilometrageMax` → utilisation de `$gte` et `$lte`
- Puissance : `puissanceMin` et `puissanceMax` → utilisation de `$gte` et `$lte`

Champs numériques exacts :

- Nombre de portes : correspondance exacte
- Nombre de places : correspondance exacte

Logique de construction de la requête :

1. Nettoyage des paramètres vides
2. Construction d'un objet de requête MongoDB
3. Application des filtres textuels (regex) et numériques (plages ou exact)
4. Combinaison avec l'opérateur `$and` implicite de MongoDB

1.2. Système de pagination

Paramètres :

- `page` : numéro de page (défaut : 1, minimum : 1)
- `limit` : nombre d'éléments par page (défaut : 10, minimum : 1)

Fonctionnement :

1. Calcul du total de documents correspondant aux filtres : `countDocuments(query)`
2. Calcul du nombre total de pages : `Math.ceil(total / limit)`
3. Application de `skip()` et `limit()` sur la requête
4. Retour des métadonnées de pagination :
    - `currentPage`

- `itemsPerPage`
- `totalPages`
- `totalItems`

1.3. Recherche de produits similaires

Algorithme de similarité :

Calcul d'un score de similarité entre une voiture cible et chaque voiture du catalogue.

Note : Cette fonctionnalité pourrait être retravaillée afin d'obtenir de meilleurs résultats. L'algorithme actuel utilise un système de scoring pondéré qui pourrait être optimisé avec des techniques de machine learning ou des algorithmes de recommandation plus avancés.

Pondération des critères :

- Prix : 17
- Marque : 7
- Modèle : 7
- Boîte : 7
- Kilométrage : 6
- État : 5
- Puissance : 5
- Tags : 4
- Carburant : 4
- Couleur : 3
- Portes : 3
- Places : 3

Calcul du score :

1. **Champs textuels** (marque, modèle, couleur, carburant, boîte, état) :
    - Correspondance exacte (insensible à la casse) : 1.0
    - Inclusion partielle : 0.5
    - Aucune correspondance : 0.0
    - Score = poids × coefficient
2. **Champs numériques** (année, prix, kilométrage, puissance) :
    - Ratio de proximité : `1 - (différence absolue / valeur maximale)`
    - Score = poids × ratio de proximité
3. **Champs exacts** (portes, places) :

- Correspondance exacte : poids complet
- Sinon : 0

4. **Tags** (tableau) :
   - Coefficient de Jaccard : `intersection / union`
   - Score = poids × coefficient de Jaccard

Processus :

1. Récupération de la voiture cible par ID
2. Calcul du score pour toutes les voitures
3. Tri décroissant par score
4. Sélection des 11 meilleurs résultats
5. Exclusion de la voiture cible
6. Retour des 10 voitures les plus similaires

1.4. Récupération des détails d'un véhicule

Endpoint : `GET /api/cars/:id`

Fonctionnement :

- Recherche par `_id` MongoDB
- Retour des détails complets du véhicule
- Gestion d'erreur si non trouvé

1.5. Gestion des erreurs et sécurité

Nettoyage des paramètres :

- Suppression des paramètres vides avant traitement
- Validation des types numériques
- Protection contre les valeurs invalides

CORS :

- Configuration CORS pour autoriser les requêtes depuis le frontend
- Origines autorisées : `localhost:5173` et `localhost:3000`

Performance :

- Middleware de mesure du temps de réponse
- Headers `X-Response-Time` pour le monitoring

---

2. Schéma de l'architecture de données

2.1. Structure de la base de données

Base de données : `uiux` (MongoDB)

Type de base de données : Base de données non relationnelle (NoSQL)

Justification du choix MongoDB :

- **Volume de données** : MongoDB est adapté à la gestion d'un grand volume de données, ce qui est essentiel pour un site de vente de véhicules avec un catalogue important
- **Intégration JavaScript** : Excellente intégration avec JavaScript/TypeScript, permettant un développement full-stack cohérent et une manipulation directe des données JSON
- **Flexibilité du schéma** : Permet d'ajouter facilement de nouveaux champs sans migration complexe, utile pour l'évolution du catalogue
- **Performance** : Optimisé pour les requêtes de lecture fréquentes sur de grandes collections

Collections principales :

- `car` : catalogue de véhicules (collection principale actuellement utilisée)
- `user` : utilisateurs de la plateforme

2.2. Collection `car`

Schéma Mongoose :

Champ	Type	Contraintes	Description
`_id`	ObjectId	Auto-généré, unique	Identifiant MongoDB
`id`	String	Requis, unique	Identifiant métier
`marque`	String	Requis, trim	Marque du véhicule
`modele`	String	Requis, trim	Modèle du véhicule
`annee`	Number	Requis, min: 1886	Année de fabrication
`prix`	Number	Requis, min: 1000	Prix en euros
`couleur`	String	Requis, trim	Couleur principale
`kilometrage`	Number	Requis, default: 0, min: 0	Kilométrage en km
`carburant`	String	Requis, trim	Type de carburant (essence, diesel, électrique, hybride)

Champ	Type	Contraintes	Description
puissance	Number	Requis, min: 0	Puissance en chevaux
boite	String	Requis, trim	Type de boîte (manuelle, automatique)
portes	Number	Requis, min: 0	Nombre de portes
places	Number	Requis, min: 0	Nombre de places
description	String	Requis, trim	Description détaillée
tags	Array[String]	Requis, default: []	Tags de catégorisation
état	String	Requis, enum: ["occasion", "neuve"]	État du véhicule
image	String	Requis, trim	URL de l'image
createdAt	Date	Auto-généré	Date de création (timestamps)
updatedAt	Date	Auto-généré	Date de modification (timestamps)

Définition du schéma (code Mongoose) :

```
const carSchema = new mongoose.Schema(
  {
    id: { type: String, required: true, unique: true },
    marque: { type: String, required: true, trim: true },
    modele: { type: String, required: true, trim: true },
    annee: { type: Number, required: true, min: 1886 },
    prix: { type: Number, required: true, min: 1000 },
    couleur: { type: String, required: true, trim: true },
    kilometrage: { type: Number, required: true, default: 0, min: 0 },
    carburant: { type: String, required: true, trim: true },
    puissance: { type: Number, required: true, min: 0 },
    boite: { type: String, required: true, trim: true },
    portes: { type: Number, required: true, min: 0 },
    places: { type: Number, required: true, min: 0 },
    description: { type: String, required: true, trim: true },
    tags: { type: [String], default: [], required: true },
    état: { type: String, enum: ["occasion", "neuve"], required: true },
    image: { type: String, required: true, trim: true },
  },
  { timestamps: true }
);
```

Index recommandés :

- Index sur `id` (unique)
- Index sur `marque`, `modele` (pour la recherche)
- Index sur `prix`, `annee`, `kilometrage` (pour les filtres de plage)
- Index composé sur les critères de filtrage fréquents

2.3. Collection `user`

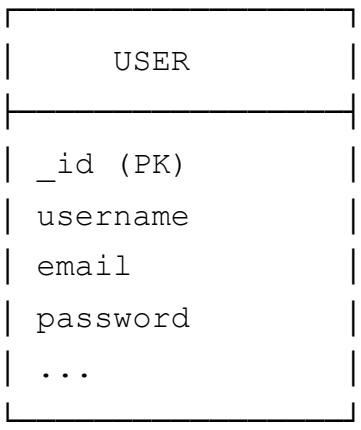
Structure (basée sur users.json) :

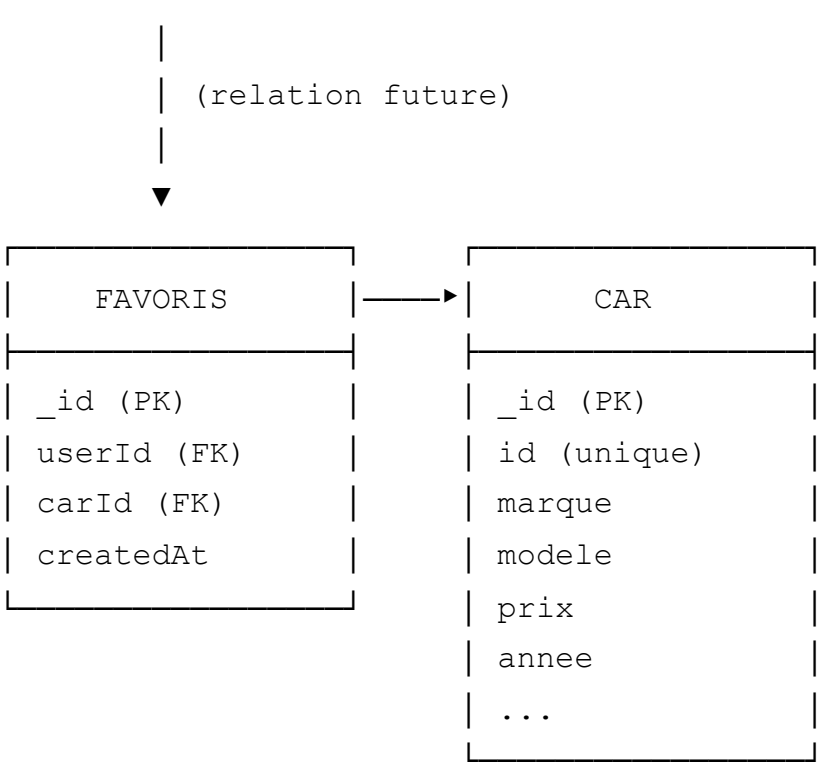
Champ	Type	Description
`_id`	ObjectId	Identifiant MongoDB
`username`	String	Nom d'utilisateur unique
`firstname`	String	Prénom
`lastname`	String	Nom
`email`	String	Adresse email
`password`	String	Mot de passe hashé (bcrypt)
`avatar`	Object	`{ url: String }`
`phonenumber`	String	Numéro de téléphone

Relations potentielles (non implémentées actuellement) :

- `user` → `favoris` : collection de références vers des véhicules
- `user` → `annonces` : véhicules publiés par l'utilisateur

2.4. Diagramme de relations





2.5. Exemples de requêtes MongoDB

Filtrage par prix :

```
db.car.find({
  prix: { $gte: 10000, $lte: 20000 },
});
```

Recherche textuelle (marque) :

```
db.car.find({
  marque: { $regex: "Peugeot", $options: "i" },
});
```

Filtrage combiné :

```
db.car.find({
  marque: { $regex: "Renault", $options: "i" },
  prix: { $gte: 15000, $lte: 25000 },
  carburant: "essence",
  portes: 5,
});
```

Pagination :

```
db.car
  .find(query)
  .skip((page - 1) * limit)
  .limit(limit);
```

2.6. Optimisations possibles

1. **Index composés** pour les filtres fréquents :

   ```
   db.car.createIndex({ marque: 1, prix: 1, carburant: 1 });
   ```

2. **Index textuel** pour la recherche full-text :

   ```
   db.car.createIndex({
     marque: "text",
     modele: "text",
     description: "text",
   });
   ```

3. **Agrégation** pour le calcul de similarité :
   - Utiliser `$lookup` et `$addFields` pour optimiser les calculs

4. **Cache Redis** :
   - Mise en cache des résultats de recherche fréquents
   - Cache des voitures similaires par ID

---

3. Détails techniques supplémentaires

3.1. Comment le filtre par prix fonctionne côté serveur

Le filtre par prix utilise les opérateurs MongoDB `$gte` (greater than or equal) et `$lte` (less than or equal) :

1. **Réception des paramètres** : `prixMin` et `prixMax` depuis les query parameters
2. **Validation** : Conversion en nombre et vérification de la validité
3. **Construction de la requête** :
   - Si `prixMin` est fourni : `prix: { $gte: Number(prixMin) }`

    - Si `prixMax` est fourni : `prix: { $lte: Number(prixMax) }`
    - Si les deux sont fournis : `prix: { $gte: Number(prixMin), $lte: Number(prixMax) }`
4. **Exécution** : MongoDB filtre les documents selon ces critères
5. **Résultat** : Retour des véhicules dont le prix est dans la plage spécifiée

Exemple concret :

- Requête : `GET /api/cars?prixMin=15000&prixMax=25000`
- Requête MongoDB générée : `{ prix: { $gte: 15000, $lte: 25000 } }`
- Résultat : Tous les véhicules entre 15 000€ et 25 000€

3.2. Comment les résultats similaires sont générés

L'algorithme de similarité utilise un système de scoring pondéré :

1. **Récupération de la voiture cible** : Par son `_id`
2. **Récupération de toutes les voitures** : Pour comparaison
3. **Calcul du score pour chaque voiture** :
    - Pour chaque critère (marque, prix, etc.), calcul d'un score partiel
    - Multiplication par le poids du critère
    - Somme de tous les scores partiels
4. **Tri par score décroissant** : Les voitures les plus similaires en premier
5. **Sélection** : Les 11 meilleures (pour exclure la voiture cible)
6. **Filtrage** : Exclusion de la voiture cible elle-même
7. **Retour** : Les 10 voitures les plus similaires

Exemple de calcul :

- Voiture cible : Peugeot 208, 2022, 18 900€, essence, 25 000 km
- Voiture comparée : Peugeot 208, 2021, 19 500€, essence, 30 000 km
- Score marque (exact) : 7 × 1.0 = 7
- Score prix (proximité) : 17 × (1 - |18900-19500|/19500) ≈ 17 × 0.97 = 16.49
- Score carburant (exact) : 4 × 1.0 = 4
- Score total ≈ 27.49

3.3. Comment la recherche est effectuée

Recherche textuelle (full-text partiel) :

Utilisation de regex MongoDB avec l'option `i` (case-insensitive) :

```
query[field] = { $regex: value, $options: "i" };
```

- **Avantages** : Recherche partielle, insensible à la casse
- **Limitations** : Pas de recherche fuzzy (typos), pas de ranking par pertinence
- **Champs concernés** : marque, modele, couleur, carburant, boite, état

Améliorations possibles :

- Implémentation d'un index textuel MongoDB pour une recherche full-text native
- Utilisation d'Elasticsearch pour une recherche avancée avec fuzzy matching
- Algorithme de Levenshtein pour la tolérance aux fautes de frappe

3.4. Organisation des données dans la base

Structure MongoDB (NoSQL) :

- **Base de données** : `uiux`
- **Collections** : Documents JSON stockés dans des collections
- **Schéma flexible** : Validation via Mongoose Schema
- **Relations** : Références par `_id` (pas de jointures automatiques)

Avantages de cette architecture :

- Flexibilité pour ajouter de nouveaux champs
- Performance pour les requêtes simples
- Scalabilité horizontale

Inconvénients :

- Pas de relations automatiques (nécessite des requêtes séparées)
- Pas de contraintes d'intégrité référentielle au niveau base

3.5. Relations entre entités

Relation actuelle :

- `car` : Collection indépendante, pas de relation explicite

Relations futures possibles :

1. **User → Favoris → Car** :

```
Collection: favoris
{
```

```
  _id: ObjectId,
  userId: ObjectId (référence vers user),
  carId: ObjectId (référence vers car),
  createdAt: Date
}
```

2. **User → Annonces → Car** :

```
Collection: car
{
  ...champs existants,
  sellerId: ObjectId (référence vers user),
  publishedAt: Date
}
```

3. **Car → Catégories** :
   - Actuellement géré via le champ `tags` (array)
   - Possibilité d'une collection séparée pour une meilleure normalisation

Modèle de données relationnel conceptuel :

```
USER (1) ─────< (N) FAVORIS (N) >─── (1) CAR
  │
  │ (1)
  │
  └─────< (N) CAR (sellerId)
```

---

4. Endpoints API

4.1. Liste des endpoints

Méthode	Endpoint	Description	Paramètres
GET	`/api/cars`	Liste des véhicules avec filtres et pagination	Query params (filtres, page, limit)
GET	`/api/cars/:id`	Détails d'un véhicule	`id` dans l'URL
GET	`/api/cars/:id/similar`	Véhicules similaires	`id` dans l'URL

4.2. Format des réponses

Liste des véhicules :

```
{
  "message": "List of cars",
  "data": [
    /* array de véhicules */
  ],
  "pagination": {
    "currentPage": 1,
    "itemsPerPage": 10,
    "totalPages": 5,
    "totalItems": 50
  }
}
```

Détails d'un véhicule :

```
{
  "message": "Details of car with id: ...",
  "data": {
    /* objet véhicule */
  }
}
```

Véhicules similaires :

```
{
  "message": "Similar cars to car with id: ...",
  "data": [
    /* array de véhicules similaires */
  ]
}
```

---

5. Fonctionnalités à venir

5.1. Système de meilleures ventes

Description :

L'outil "meilleures ventes" est envisagé et a une place présente sur le site, mais nécessite des données supplémentaires pour être fonctionnel.

Défis actuels :

- **Manque de données** : Sans une grande base de données et des utilisateurs réguliers, il est difficile de déterminer ce qu'est une "meilleure vente"
- **Métriques nécessaires** : Nécessite le suivi de :
  - Nombre de vues par véhicule
  - Nombre de contacts/demandes
  - Taux de conversion
  - Historique des ventes effectives

Implémentation future :

- Création d'une collection `analytics` ou ajout de champs dans la collection `car` :
  - `views` : nombre de vues
  - `contacts` : nombre de contacts générés
  - `sold` : booléen indiquant si le véhicule a été vendu
  - `soldAt` : date de vente
- Calcul périodique d'un score de popularité
- Endpoint API : `GET /api/cars/bestsellers`

5.2. Système de commentaires (scraping Trustpilot)

Description :

Intégration de commentaires clients via scraping du site Trustpilot pour améliorer la confiance et la transparence.

Architecture proposée :

Nouvelle collection : `comments`

```
const commentSchema = new mongoose.Schema(
  {
    _id: ObjectId,
    source: { type: String, enum: ["trustpilot"], required: true },
    author: { type: String, required: true },
    rating: { type: Number, min: 1, max: 5, required: true },
    content: { type: String, required: true },
    date: { type: Date, required: true },
    verified: { type: Boolean, default: false },
```

```
    scrapedAt: { type: Date, default: Date.now },
  },
  { timestamps: true }
);
```

Processus de scraping :

1. **Script de scraping** : Extraction périodique des commentaires depuis Trustpilot
2. **Stockage** : Enregistrement dans la collection `comments`
3. **Exposition** : Affichage rapide via endpoint dédié : `GET /api/comments`
4. **Mise à jour** : Actualisation régulière pour maintenir la fraîcheur des données

Avantages :

- Amélioration de la crédibilité du site
- Contenu riche sans intervention manuelle
- Mise à jour automatique des avis

Considérations techniques :

- Respect des conditions d'utilisation de Trustpilot
- Gestion du rate limiting
- Nettoyage et validation des données scrapées
- Cache pour optimiser les performances

5.3. Amélioration de l'algorithme de produits similaires

Limitations actuelles :

- Calcul effectué sur toutes les voitures (peut être lent avec un grand catalogue)
- Pondération fixe qui pourrait être optimisée
- Pas de prise en compte des préférences utilisateur

Améliorations envisagées :

1. **Optimisation des performances :**
    - Pré-calcul des similarités pour les véhicules les plus consultés
    - Utilisation d'index MongoDB pour limiter le scope de recherche
    - Cache des résultats de similarité
2. **Amélioration de l'algorithme :**
    - Machine learning pour ajuster les poids dynamiquement
    - Prise en compte de l'historique de navigation utilisateur

- Filtrage par catégories similaires (même segment de marché)

3. **Nouvelles métriques :**
    - Similarité basée sur les tags (coefficient de Jaccard amélioré)
    - Prise en compte de la popularité (voitures similaires populaires en priorité)
    - Filtrage par disponibilité (exclure les véhicules vendus)

Implémentation future :

- Endpoint optimisé : `GET /api/cars/:id/similar?limit=10&category=segment`
- Background job pour pré-calculer les similarités
- Intégration avec le système de recommandation utilisateur
