



# RAPPORT : RÉALISATION D'UNE API RESTFUL CONNECTÉE À MONGODB

Fait par : Abdoul Aziz BALMA

Le 07/07/2025

# **Plan**

<b><i>Plan</i></b> .....	<b>2</b>
<b><i>INTRODUCTION</i></b> .....	<b>3</b>
<b><i>I. Choix et description du jeu de données</i></b> .....	<b>4</b>
<b><i>II. Listes des Requêtes proposées dans l'API</i></b> .....	<b>7</b>
<b><i>III. Description de la procédure de déploiement de la base de données sur Mongo Atlas</i></b> .....	<b>14</b>
<b><i>IV. Présentation et explication des codes sources</i></b> .....	<b>20</b>
<b><i>V. Documentation de l'API</i></b> .....	<b>30</b>
<b><i>Conclusion</i></b> .....	<b>37</b>

# INTRODUCTION

Concevoir une API performante, flexible et sécurisée est aujourd’hui un levier stratégique dans le développement d’applications modernes orientées données.

Dans ce contexte, le présent projet a pour objectif la création d’une API RESTful en Python à l’aide du microframework Flask, couplée à une base de données MongoDB hébergée sur MongoDB Atlas, et alimentée par le jeu de données hotel\_bookings.csv, largement reconnu dans le domaine de la science des données pour son potentiel analytique.

Ce jeu de données, riche et structuré, offre une excellente base pour illustrer la création d’un service d’API capable de fournir des informations ciblées sur les réservations d’hôtel, tout en permettant des opérations de filtrage, d’analyse et d’exploration des données à travers des requêtes simples et avancées. Il s’inscrit également dans une logique pédagogique visant à démontrer comment une architecture Flask bien pensée peut dialoguer avec un système NoSQL (MongoDB) pour offrir une API rapide, flexible et extensible.

Au-delà de la simple exposition de données, ce projet met l’accent sur la structuration des routes, la documentation de l’API, la gestion efficace des données via MongoDB Atlas, ainsi que sur la mise en œuvre de tests pour en valider la robustesse.

À travers une démarche progressive, ce rapport détaille les différentes étapes du projet : depuis le choix et l’importation du jeu de données, en passant par la conception des routes de l’API, jusqu’à la mise en production et la validation finale via des captures de tests. L’ensemble vise à fournir une solution professionnelle, maintenable et évolutive, alignée avec les bonnes pratiques du développement d’API modernes.

## I. Choix et description du jeu de données

Dans le cadre de ce projet, nous avons choisi d'exploiter le jeu de données **hotel\_bookings.csv**, un fichier riche, réel et largement reconnu dans le domaine de la data science sur **Kaggle**. Ce dataset contient **119 391 enregistrements** correspondant à des réservations effectuées dans deux types d'établissements hôteliers : un **hôtel urbain (City Hotel)** et un **hôtel de type resort (Resort Hotel)**.

Chaque ligne représente une **réservation unique**, incluant des données sur le client, les modalités de séjour, les informations financières et le statut final de la réservation. Ces données sont précieuses pour des applications telles que la prévision de la demande, l'analyse de la satisfaction client, la détection des comportements à risque (ex. : annulations multiples), ou encore l'optimisation des revenus.

← → ⌛ kaggle.com/datasets/jessemstipak/hotel-booking-demand/data

Search

JESSE MOSTIPAK - UPDATED 5 YEARS AGO

▲ 2562 ◀ ▶ Code Download ⋮

## Hotel booking demand

From the paper: hotel booking demand datasets

Data Card Code (572) Discussion (28) Suggestions (0)

### About Dataset

**Usability** 10.00

**License** Attribution 4.0 International (CC ...

**Expected update frequency**



← → ⌛ kaggle.com/datasets/jessemstipak/hotel-booking-demand/data

Search

JESSE MOSTIPAK - UPDATED 5 YEARS AGO

▲ 2562 ◀ ▶ Code Download ⋮

## Hotel booking demand

Data Card Code (572) Discussion (28) Suggestions (0)

hotel	# is_canceled	# lead_time	# arrival_date_year	# arrival_date_month	# arriva
Hotel (H1 = Resort Hotel or H2 = City Hotel)	Value indicating if the booking was canceled (1) or not (0)	Number of days that elapsed between the entering date of the booking into the PMS and the arrival date	Year of arrival date	Month of arrival date	Week number of arrival date
City Hotel 66%	0	1	2015	2017	1
Resort Hotel 34%	0	737	2015	2017	1
Resort Hotel	0	342	2015	July	27
Resort Hotel	0	737	2015	July	27
Resort Hotel	0	7	2015	July	27
Resort Hotel	0	13	2015	July	27

**Summary**

- 1 file
- 32 columns

## Pourquoi ce choix de dataset ?

Le choix de ce jeu de données repose sur plusieurs points forts :

- **Volume conséquent (119 391 lignes)** permettant des requêtes et agrégations significatives
- **Diversité des données** favorisant des traitements analytiques complexes (statistiques, filtres avancés, regroupements)
- **Format bien structuré** facilement intégrable dans une base **MongoDB Atlas**, avec un excellent rendu en documents JSON

**Utilisation concrète** dans des cas réels de gestion hôtelière : suivi des annulations, analyse de revenus, profils clients ...

## Intérêt pour l'API

Grâce à cette richesse, nous avons pu concevoir une API RESTful complète capable de traiter :

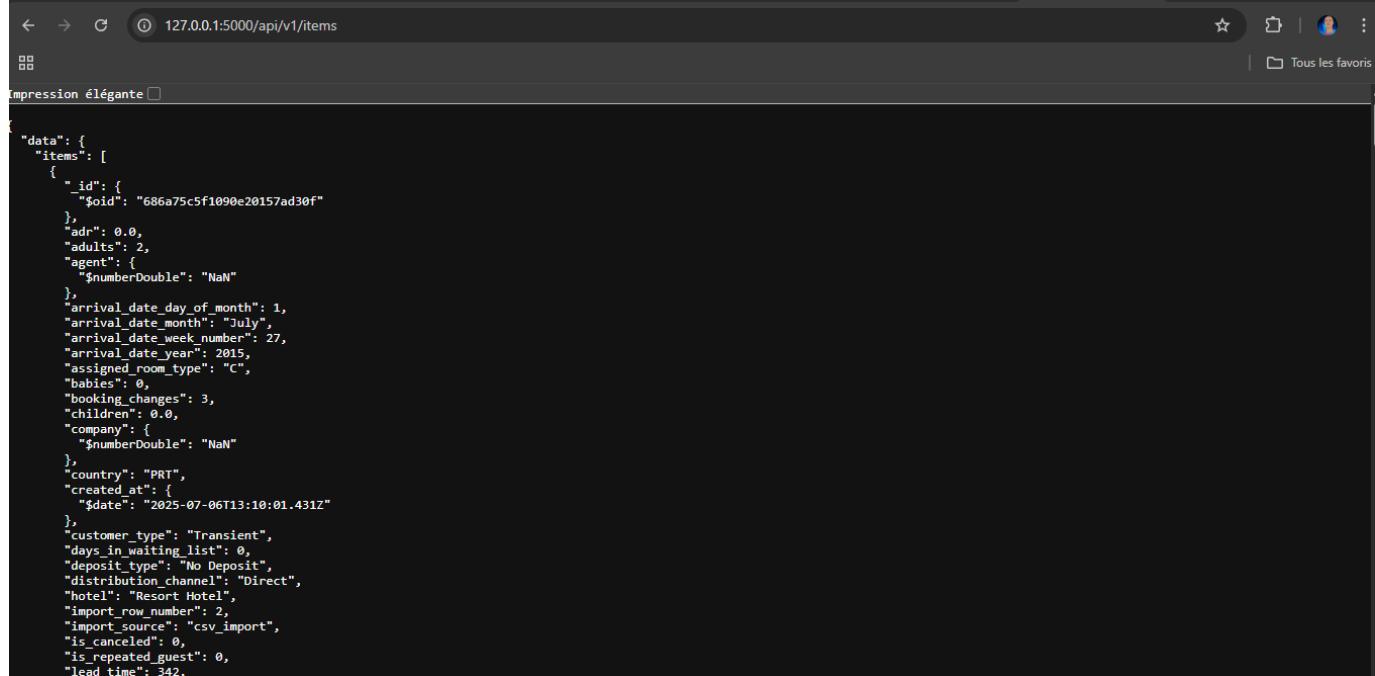
- Des **requêtes simples** (afficher les réservations par pays ou par hôtel),
- Des **requêtes complexes** (calcul du taux d'annulation par mois, moyenne des revenus selon les segments, identification de clients fidèles...),
- Tout en gardant une **logique évolutive**, utile pour une intégration dans des systèmes d'analyse ou des tableaux de bord métier.

Ce dataset constitue donc une base idéale pour démontrer les capacités d'une API performante, connectée à une base de données NoSQL, tout en répondant à des besoins métiers concrets.

## II. Listes des Requêtes proposées dans l'API

### 1. GET /api/v1/items

- **Description :** Récupère tous les documents de la collection contenant les réservations hôtelières.
- **Exemple de Requête :**  
GET <http://127.0.0.1:5000/api/v1/items>
- **Réponse :**
  - **Code 200 :** Succès. Retourne un tableau JSON contenant tous les documents.
  - **Note :** La réponse peut être paginée selon l'implémentation exemple d'ici.



The screenshot shows a browser window with the URL `127.0.0.1:5000/api/v1/items` in the address bar. The page content is a JSON object with the following structure:

```
{ "data": { "items": [ { "_id": { "$oid": "686a75c5f1090e20157ad30f" }, "adr": 0.0, "adults": 2, "agent": { "$numberDouble": "NaN" }, "arrival_date_day_of_month": 1, "arrival_date_month": "July", "arrival_date_week_number": 27, "arrival_date_year": 2015, "assigned_room_type": "C", "babies": 0, "booking_changes": 3, "children": 0.0, "company": { "$numberDouble": "NaN" }, "country": "PRT", "created_at": { "$date": "2025-07-06T13:10:01.431Z" }, "customer_type": "Transient", "days_in_waiting_list": 0, "deposit_type": "No Deposit", "distribution_channel": "Direct", "hotel": "Resort Hotel", "import_row_number": 2, "import_source": "csv_import", "is_canceled": 0, "is_repeated_guest": 0, "lead_time": 342, } ] } }
```

### 2. GET /api/v1/items/<id>

- **Description :** Récupère un document spécifique par son identifiant unique (`_id`) dans la base de données MongoDB.

- **Exemple de Requête :**

GET http://127.0.0.1:5000/api/v1/items/686a75c5f1090e20157ad30f

- **Réponse :**

- **Code 200** : Succès. Retourne le document correspondant.
- **Code 404** : Non trouvé. Si l'ID ne correspond à aucun document.
- **Code 500** : Erreur serveur.

The screenshot shows the Thunder Client interface. In the top bar, there's a search field with 'flask\_api'. Below it, a 'New Request' button is highlighted. The main area shows a 'Query' tab selected, with the URL '127.0.0.1:5000/api/v1/items/686a75c5f1090e20157ad30f' and a 'Send' button. To the right, the response details are shown: 'Status: 200 OK', 'Size: 1.28 KB', and 'Time: 133 ms'. The response body is a JSON object:

```

1  {
2    "data": {
3      "_id": {
4        "$oid": "686a75c5f1090e20157ad30f"
5      },
6      "adr": 0.0,
7      "adults": 2,
8      "agent": {
9        "$numberDouble": "NaN"
10     },
11     "arrival_date_day_of_month": 1,
12     "arrival_date_month": "July",
13     "arrival_date_week_number": 27,
14     "arrival_date_year": 2015,
15     "assigned_room_type": "C",
16     "babies": 0,
17     "booking_changes": 3,
18     "children": 0.0,
19     "company": {
20       "$numberDouble": "NaN"

```

At the bottom, the terminal shows the command 'curl -X GET http://127.0.0.1:5000/api/v1/items/686a75c5f1090e20157ad30f' and its output.

### 3. GET /api/v1/items/country/<country\_code>

- **Description** : Récupère toutes les réservations faites par des clients originaires d'un pays donné (ex : PRT, FRA, USA...).

- **Exemple de Requête :**

GET http://127.0.0.1:5000/api/v1/items/country/USA

The screenshot shows a Postman interface with the following details:

- Request URL:** GET http://127.0.0.1:5000/api/v1/items/country/USA
- Status:** 200 OK
- Size:** 1.67 MB
- Time:** 3.21 s
- Response Content:**

```

1  {
2   "data": [
3     "items": [
4       {
5         "_id": {
6           "$oid": "686a75c5f1090e20157ad31b"
7         },
8         "adr": 97.0,
9         "adults": 2,
10        "agent": 240.0,
11        "arrival_date_day_of_month": 1,
12        "arrival_date_month": "July",
13        "arrival_date_week_number": 27,
14        "arrival_date_year": 2015,
15        "assigned_room_type": "E",
16        "babies": 0,
17        "booking_changes": 0,
18        "children": 0.0,
19        "company": {
20          "$numberDouble": "NaN"
21        },
22        "country": "USA",
23        "created_at": {
24          "$date": "2025-07-06T13:10:01.431Z"
25        },
26        "customer_type": "Transient"
      }
    ]
  }

```

- **Réponse :**
  - **Code 200** : Succès. Retourne un tableau filtré.
  - **Code 404** : Aucun document trouvé.

#### 4. POST /api/v1/items

- **Description :** Crée un nouveau document (réservation) dans la base de données.
- **Exemple de Requête :**  
POST **http://127.0.0.1:5000/api/v1/items**
- **Body (JSON) :**

```
{  
    "hotel": "City Hotel",  
    "country": "FRA",  
    "adults": 2,  
    "children": 0,  
    "arrival_date_month": "October",  
    "reservation_status": "Check-Out",  
    "adr": 145.50  
}
```

The screenshot shows the Postman interface with a successful API call. The status bar at the top indicates "Status: 201 CREATED Size: 473 Bytes Time: 242 ms". The response body is a JSON object:

```

1  {
2    "data": {
3      "_id": {
4        "$oid": "686b21efbb01a6b5ca6368ae"
5      },
6      "adr": 145.5,
7      "adults": 2,
8      "arrival_date_month": "October",
9      "children": 0,
10     "country": "FRA",
11     "created_at": {
12       "$date": "2025-07-07T01:25:03.126Z"
13     },
14     "hotel": "City Hotel",
15     "reservation_status": "Check-Out",
16     "updated_at": {
17       "$date": "2025-07-07T01:25:03.126Z"
18     }
19   },
20   "message": "Item créé avec succès",

```

The terminal section shows log entries indicating the creation of the item.

## Réponse :

- Code 201** : Succès. Retourne l'`_id` du document inséré.
- Code 400** : Données manquantes ou mal formatées.
- Code 500** : Erreur serveur.

## 5. PUT /api/v1/items/<id>

- Description** : Met à jour un document existant selon son identifiant MongoDB.
  - Exemple** :
- PUT  
<http://127.0.0.1:5000/api/v1/items/686a75c5f1090e20157ad30f>
- Body (JSON)** :

```
{
}
```

```

"adr": 110.75,
"reservation_status": "Canceled"
}

```

The screenshot shows the Postman interface with the following details:

- Method:** PUT
- URL:** http://127.0.0.1:5000/api/v1/items/686a75c5f1090e20157ad30f
- Body:** JSON content (circled in red) sent via the Body tab.
- Response Headers:** Status: 200 OK, Size: 1.33 KB, Time: 415 ms
- Response Body:** A detailed JSON object representing the updated item, including fields like \_id, adr, adults, agent, arrival\_date, assigned\_room\_type, babies, booking\_changes, children, company, and more.
- Terminal:** Shows log entries indicating the item was updated and the request was successful (HTTP/1.1 200).

## Réponse :

- **Code 200** : Succès. Données mises à jour.
- **Code 404** : Document non trouvé.
- **Code 400** : Données invalides.
- **Code 500** : Erreur serveur.

## 6. DELETE /api/v1/items/<id>

- **Description :** Supprime un document par son identifiant.

- **Exemple :**

DELETE

**http://127.0.0.1:5000/api/v1/items/ 686a75c5f1090e20157ad310"**

The screenshot shows the Postman interface with a successful API call. The top bar says "TC New Request". The request method is "DELETE" and the URL is "http://127.0.0.1:5000/api/v1/items/686a75c5f1090e20157ad310". The response status is circled in red as "200 OK". The response body is:

```

1 {
2   "message": "Item supprimé avec succès",
3   "status": "success"
4 }

```

The bottom section shows terminal logs:

```

2025-07-07 01:40:32,404 INFO models.item Item 686a75c5f1090e20157ad310 supprimé
2025-07-07 01:40:32,405 INFO werkzeug 127.0.0.1 - - [07/Jul/2025 01:40:32] "DELETE /api/v1/items/686a75c5f1090e20157ad310 HTTP/1.1" 200 -

```

- **Réponse :**

- **Code 200 :** Succès. Document supprimé.
- **Code 404 :** Document introuvable.
- **Code 500 :** Erreur serveur.

### III. Description de la procédure de déploiement de la base de données sur Mongo Atlas

Dans le cadre du projet **flask\_api**, la base de données a été déployée sur **MongoDB Atlas**, une solution cloud proposée par MongoDB permettant d'héberger des bases de données **NoSQL** de manière **sécurisée, évolutive et accessible à distance**.

Cette approche facilite l'intégration avec l'API Flask et garantit une meilleure accessibilité depuis n'importe quel environnement de déploiement (local ou cloud).

#### 1. Création d'un compte MongoDB Atlas

La première étape consiste à créer un compte gratuit sur la plateforme MongoDB Atlas :

- . Se rendre sur le site <https://www.mongodb.com/cloud/atlas>
- . Cliquer sur "Try Free" et suivre les étapes d'inscription.
- . Une fois connecté, accéder au **Tableau de bord**.

The screenshot shows the MongoDB Atlas dashboard with the following details:

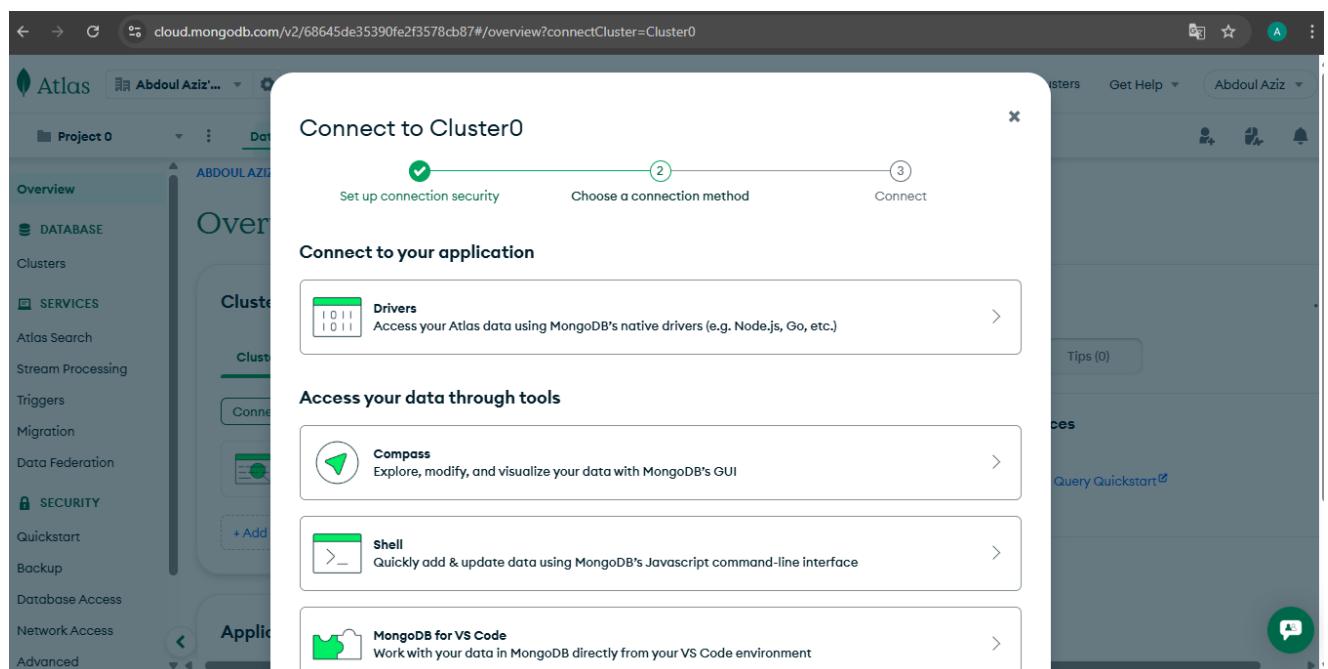
- Header:** Shows the URL [cloud.mongodb.com/v2/68645de35390fe2f3578cb87#/overview](https://cloud.mongodb.com/v2/68645de35390fe2f3578cb87#/overview), a user profile for "Abdoul Aziz...", and navigation icons.
- Left Sidebar:** Lists categories such as Project 0, Data Services (selected), Charts, Overview, DATABASE, SERVICES, SECURITY, and various management options like Access Manager and Billing.
- Central Content:**
  - Overview Section:** Displays a message: "Your organization does not have a designated security contact. Add an Atlas Security Contact in [Organization Settings](#) to receive security-related notifications." It also shows "ABDOUL AZIZ'S ORG - 2025-07-01 > PROJECT 0".
  - Clusters Section:** Shows "Cluster0" with "Data Size: 210.08 MB". Buttons for "Create cluster", "Connect", and "Edit configuration" are available.
  - Actions:** Buttons for "Browse collections" and "View monitoring".
  - Tags:** A button for "+ Add Tag".
- Right Sidebar:**
  - Toolbar:** Buttons for "Resources (6)" and "Tips (0)".
  - Featured Resources:** Categories like PYTHON, with links to "Python Connect and Query Quickstart" and "Build GenAI Apps".
  - Sample Apps:** A section with a green speech bubble icon.

## 2. Création d'un cluster de base de données

Un cluster représente un groupe de serveurs MongoDB qui hébergeront la base de données.

Étapes :

- . Cliquer sur "**Build a Database**"
- . Choisir l'option **Free (M0 Shared Cluster)** pour un projet gratuit.
- . Sélectionner un provider cloud (ex: AWS) et une région géographique proche.
- . Nommer le cluster ou laisser le nom par **défaut (Cluster0)**
- . Cliquer sur **Create Cluster**



## 3. Création d'un utilisateur de base de données

Pour permettre à l'API Flask de se connecter en toute sécurité à la base, il est nécessaire de créer un utilisateur d'accès :

- . Aller dans **Database Access**
- . Cliquer sur **Add New Database User**
- . Saisir un nom d'utilisateur et un mot de passe fort (**api\_user**, **motdepasseAPI**)
- . Donner le rôle **Read and Write to any database**
- . Valider

Ces identifiants seront utilisés dans l'URI de connexion.

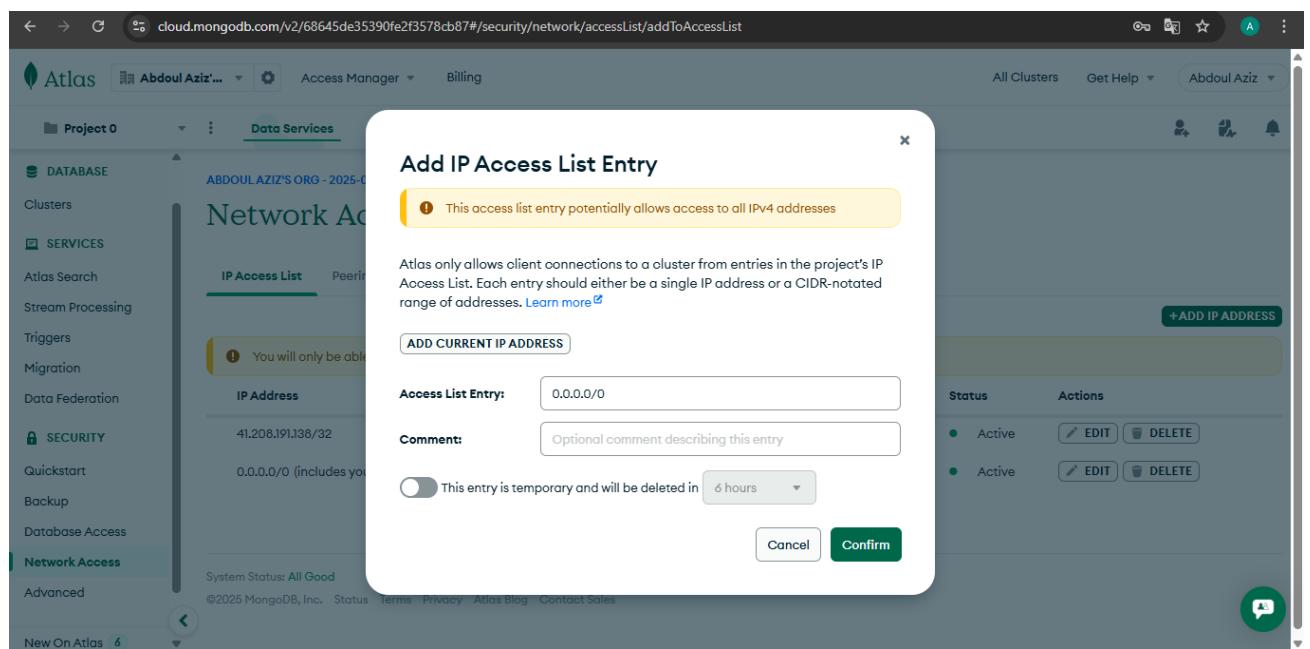
The screenshot shows the MongoDB Cloud Atlas interface for adding a new database user. The user 'api\_user' has been entered, and the password field is masked. A note about user description and database privileges is visible. The 'Built-in Role' section shows '0 SELECTED' roles, and the 'Custom Roles' section indicates pre-defined custom roles can be selected.

The screenshot shows the MongoDB Cloud Atlas interface for managing database access. The 'Database Users' tab is selected, listing the user 'api\_user' with details like authentication method (SCRAM) and MongoDB roles (atlasAdmin@admin).

## 4. Configuration des autorisations réseau (IP Whitelist)

Par défaut, MongoDB Atlas bloque les connexions extérieures. Il faut donc autoriser les adresses IP souhaitées :

- . Aller dans **Network Access**
- . Cliquer sur **Add IP Address**
- . Choisir :
  - o **0.0.0.0/0** pour autoriser l'accès depuis n'importe quelle IP (option large mais utile en développement)
  - o Ou ajouter uniquement votre **IP actuelle** pour plus de sécurité
- . Valider

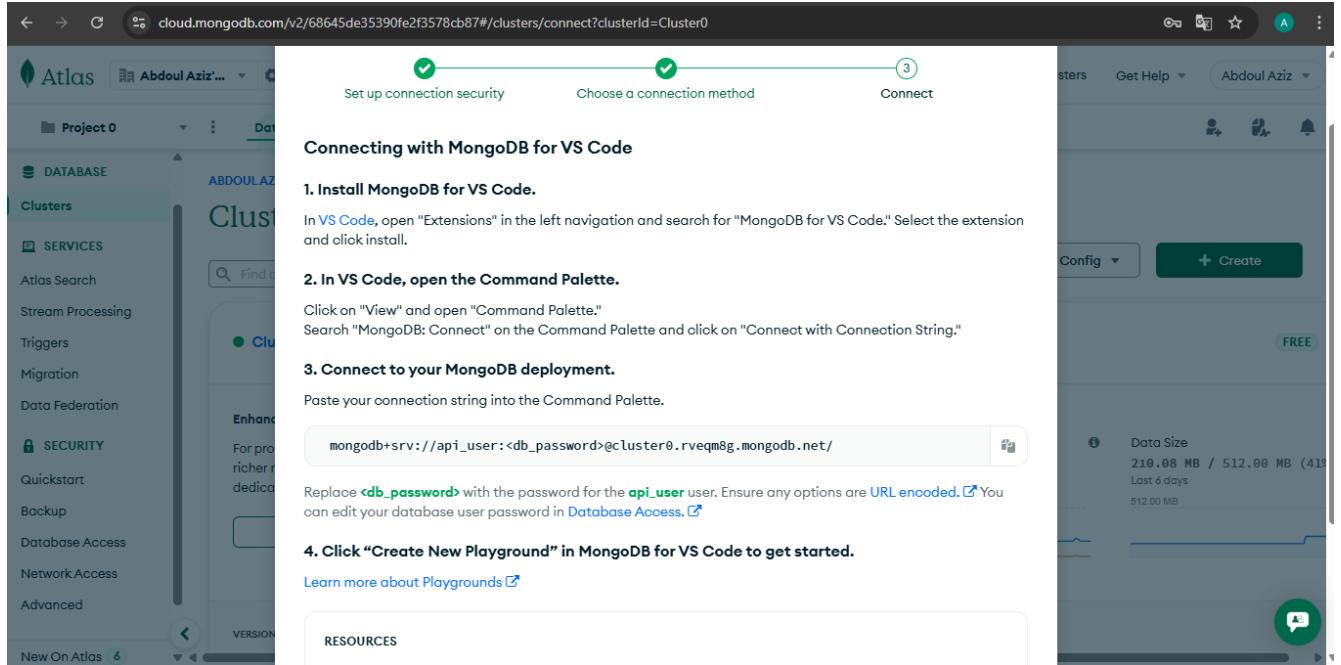


## 5. Connexion à la base de données

Une fois le cluster prêt, on peut récupérer l'URI de connexion :

- . Aller dans **Clusters**

- . Cliquer sur **Connect**
- . Choisir **Connect your application**
- . Copier l'URI standard comme :



## 6. Intégration dans l'application Flask

Le lien de connexion est ensuite stocké dans un fichier **.env** :

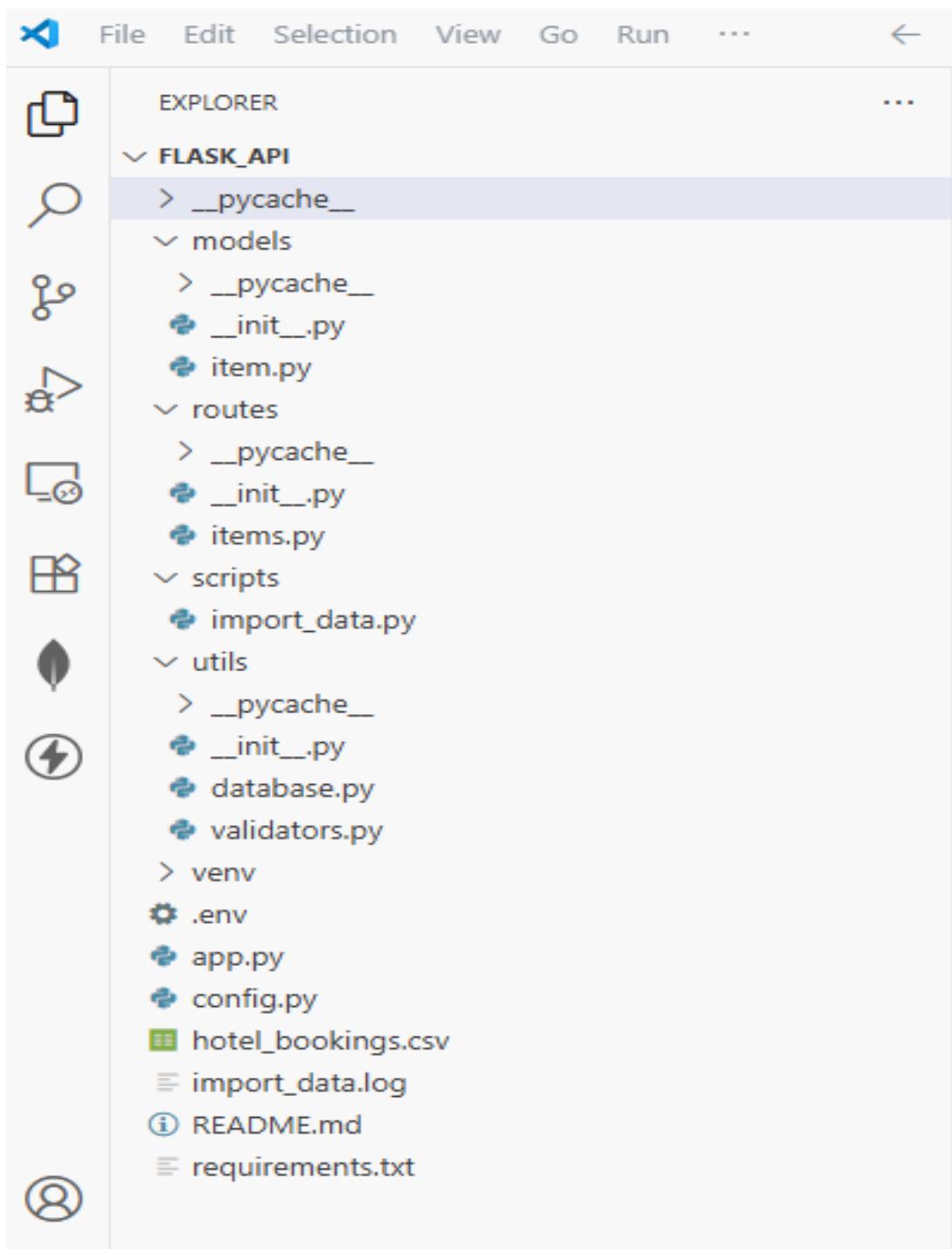
```
⚙ .env
1 # Configuration MongoDB Atlas
2 MONGO_URI=mongodb+srv://api_user:BRMQuXXXXXXXXXXIfA@cluster0.rveqm8g.mongodb.net/
3 DATABASE_NAME=mydatabase
4 COLLECTION_NAME=items
5
6 # Configuration Flask
7 FLASK_ENV=development
8 FLASK_CONFIG=development
9 SECRET_KEY=your-secret-key-here
10
11 # Configuration de sécurité (pour production)
12 # FLASK_ENV=production
13 # FLASK_CONFIG=production
14 # SECRET_KEY=your-very-secure-secret-key-for-production
15
16 # Configuration des logs
17 LOG_LEVEL=INFO
```

## **IV. Présentation et explication des codes sources**

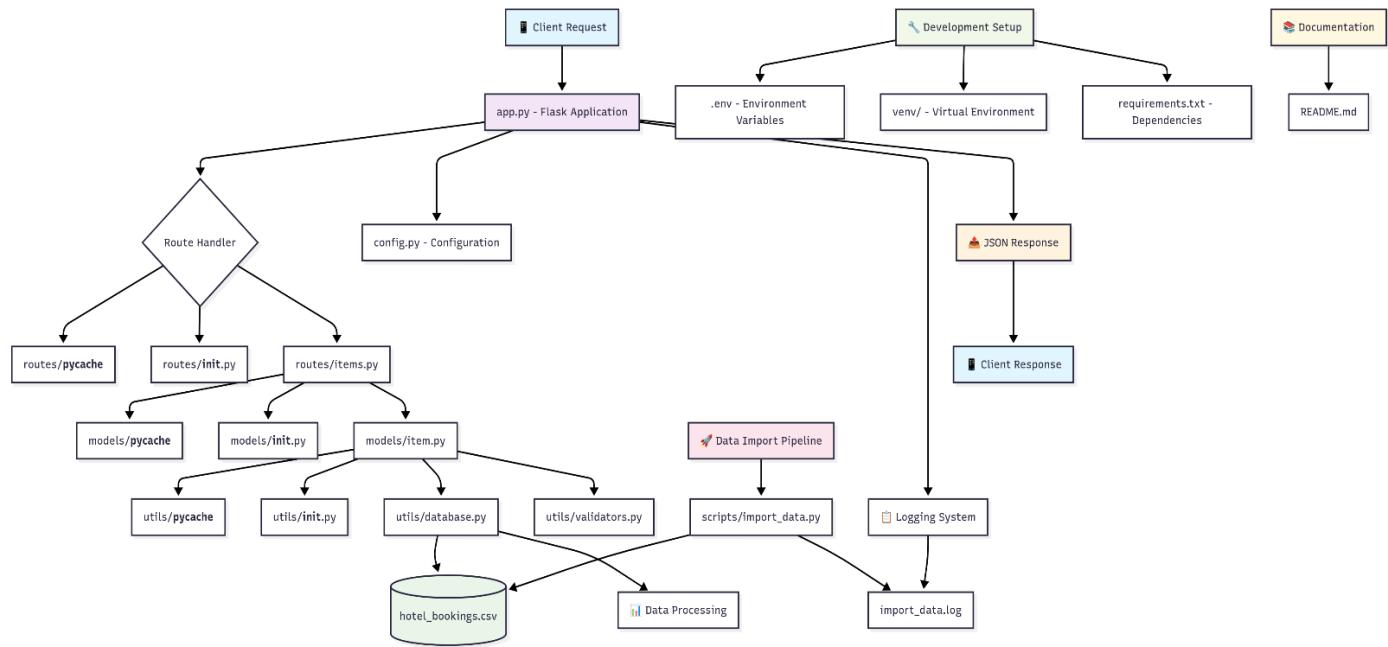
Dans cette section, nous présentons la structure de l'API développée avec Flask et MongoDB, ainsi qu'une explication claire des fichiers principaux constituant le projet flask\_api.

### **1. Arborescence générale du projet**

L'organisation du projet suit une architecture modulaire afin de garantir une meilleure lisibilité, maintenabilité et évolutivité :



## Pipeline du projet flask\_api :



```

flask_api/
|
├── app.py                                # Point d'entrée principal de l'application Flask
├── routes/
│   └── item_routes.py                      # Fichier contenant les routes relatives aux items
├── models/
│   └── item.py                            # Définition du modèle de données "Item"
├── utils/
│   ├── database.py                         # Connexion à la base MongoDB
│   ├── validators.py                       # Validation des données
│   └── logger.py                           # Gestion des logs (facultatif)
├── .env                                     # Variables d'environnement (ex: URI MongoDB)
├── requirements.txt                         # Liste des bibliothèques Python nécessaires
└── README.md                               # Documentation du projet

```

## 2. app.py – Lancement de l’application

Le fichier app.py constitue le **point d’entrée de l’application**. Il initialise l’objet Flask, configure les extensions et enregistre les routes définies dans un blueprint.

```
⌚ app.py
1  from flask import Flask, jsonify
2  from flask_cors import CORS
3  import logging
4  import os
5  from config import config, Config
6  from utils.database import db_manager
7  from routes.items import items_bp
8
9  def create_app(config_name=None):
10     """Factory function pour créer l'application Flask"""
11
12     # Créer l'application Flask
13     app = Flask(__name__)
14
15     # Configuration
16     config_name = config_name or os.getenv('FLASK_CONFIG', 'default')
17     app.config.from_object(config[config_name])
18
19     # Valider la configuration
20     try:
21         Config.validate_config()
22     except ValueError as e:
23         app.logger.error(f"Erreur de configuration: {str(e)}")
24         raise
25
26     # Configuration CORS
```

## 3. routes/item\_routes.py – Définition des routes

Ce fichier contient l’ensemble des **routes de l’API liées aux objets Item**, organisées par méthode HTTP :

```

routes > items.py
 1  from flask import Blueprint, request, jsonify, current_app
 2  from bson import json_util
 3  from models.item import Item
 4  from utils.validators import Validator, ValidationError
 5  import logging
 6  import json
 7
 8  logger = logging.getLogger(__name__)
 9  items_bp = Blueprint('items', __name__)
10
11 def create_response(data=None, message=None, status_code=200):
12     response = {}
13     if data is not None:
14         response['data'] = data
15     if message:
16         response['message'] = message
17     response['status'] = 'success' if status_code < 400 else 'error'
18     return jsonify(response), status_code
19
20 def parse_json_response(data):
21     return json.loads(json_util.dumps(data))
22
23 @items_bp.route('/items', methods=['GET'])
24 def get_items():
25     try:
26         page = request.args.get('page', '1')

```

## 4. utils/database.py – Connexion à la base MongoDB

Ce fichier configure la **connexion à la base de données MongoDB**, grâce au module pymongo et à une variable d'environnement sécurisée :

```

utils > database.py
1  from pymongo import MongoClient
2  from pymongo.errors import ServerSelectionTimeoutError, ConnectionFailure
3  import logging
4  from config import Config
5
6  logger = logging.getLogger(__name__)
7
8  class DatabaseManager:
9      """Gestionnaire de connexion à MongoDB"""
10
11     def __init__(self):
12         self.client = None
13         self.db = None
14         self.collection = None
15
16     def connect(self):
17         """Établit la connexion à MongoDB"""
18         try:
19             self.client = MongoClient(
20                 Config.MONGO_URI,
21                 serverSelectionTimeoutMS=5000,
22                 connectTimeoutMS=10000,
23                 socketTimeoutMS=20000
24             )
25
26             # Test de la connexion

```

## 5. models/item.py – Définition du modèle Item

Ce fichier contient la classe Item, représentant la **structure d'un objet manipulé par l'API**. Elle encapsule les opérations principales liées à un document MongoDB.

```

models > item.py
 1  from datetime import datetime
 2  from typing import Dict, Any, Optional, List
 3  from bson.objectid import ObjectId
 4  from utils.database import db_manager
 5  from utils.validators import Validator, ValidationError
 6  import logging
 7
 8  logger = logging.getLogger(__name__)
 9
10 class Item:
11     """Modèle pour représenter un item"""
12
13     def __init__(self, data: Dict[str, Any]):
14         self.data = data
15         self._id = data.get('_id')
16
17     @classmethod
18     def find_all(cls, page: int = 1, per_page: int = 10, filters: Optional[Dict] = None) -> Dict[str]:
19         """Récupère tous les items avec pagination et filtres"""
20         try:
21             collection = db_manager.get_collection()
22             query = filters or {}
23             skip = (page - 1) * per_page
24             cursor = collection.find(query).skip(skip).limit(per_page)
25             items = list(cursor)
26             total = collection.count_documents(query)

```

## 6. utils/validators.py – Validation des données

Ce module permet de **vérifier la validité des données** avant leur enregistrement en base, afin d'éviter les erreurs ou incohérences :

```
utils > 📁 validators.py
1   from bson.objectid import ObjectId
2   from bson.errors import InvalidID
3   import re
4   from typing import Dict, Any, List, Optional
5
6   class ValidationError(Exception):
7       """Exception personnalisée pour les erreurs de validation"""
8       pass
9
10  class Validator:
11      """Classe pour valider les données"""
12
13      @staticmethod
14      def validate_object_id(object_id: str) -> bool:
15          """Valide un ObjectId MongoDB"""
16          try:
17              ObjectId(object_id)
18              return True
19          except (InvalidID, TypeError):
20              return False
21
22      @staticmethod
23      def validate_required_fields(data: Dict[str, Any], required_fields: List[str]) -> None:
24          """Valide que les champs requis sont présents"""
25          missing_fields = [field for field in required_fields if field not in data or data[field] is None]
26
```

## 7. .env – Configuration sécurisée

Le fichier **.env** contient des **informations sensibles** (ex : URI de MongoDB) qui ne doivent jamais être exposées publiquement :

---

```
⚙ .env
1 # Configuration MongoDB Atlas
2 MONGO_URI=mongodb+srv://api_user:BRMQuXXXXXXXXXXIfA@cluster0.rveqm8g.mongodb.net/
3 DATABASE_NAME=mydatabase
4 COLLECTION_NAME=items
5
6 # Configuration Flask
7 FLASK_ENV=development
8 FLASK_CONFIG=development
9 SECRET_KEY=your-secret-key-here
10
11 # Configuration de sécurité (pour production)
12 # FLASK_ENV=production
13 # FLASK_CONFIG=production
14 # SECRET_KEY=your-very-secure-secret-key-for-production
15
16 # Configuration des logs
17 LOG_LEVEL=INFO
```

---

## 8. requirements.txt – Liste des dépendances

Ce fichier contient toutes les **bibliothèques nécessaires** au bon fonctionnement du projet :

```
☰ requirements.txt
1  # Framework Flask
2  Flask==2.3.3
3  Flask-CORS==4.0.0
4
5  # Base de données MongoDB
6  pymongo==4.6.0
7
8  # Traitement des données
9  pandas==2.1.4
10 numpy==1.26.4
11
12 # Variables d'environnement
13 python-dotenv==1.0.0
14
15 # Logs et monitoring
16 python-json-logger==2.0.7
17
18 # Tests (optionnel)
19 pytest==7.4.3
20 pytest-flask==1.3.0
21 pytest-cov==4.1.0
22
23 # Développement (optionnel)
24 black==23.12.0
25 flake8==6.1.0
26 isort==5.13.2
```

## V. Documentation de l'API

L'API **flask\_api** est une **API RESTful** conçue pour gérer des données de réservations hôtelières extraites du jeu de données public **hotel\_bookings.csv** disponible sur Kaggle.

Elle est développée en **Python avec le microframework Flask**, et s'appuie sur **MongoDB Atlas** pour la base de données. Cette API permet des opérations **CRUD (Create, Read, Update, Delete)** et expose des points d'accès personnalisés pour une exploitation précise des données.

### 1. Jeu de données utilisé

- **Nom** : hotel\_bookings.csv
- **Source** : Kaggle ([lien](#))
- **Colonnes exploitées dans l'API :**
  - hotel : type d'hôtel (City Hotel, Resort Hotel)
  - arrival\_date\_month : mois d'arrivée
  - adults, children : nombre d'adultes et d'enfants
  - country : code pays ISO (PRT, FRA, etc.)
  - booking\_status : statut de la réservation (Canceled, Not Canceled)
  - adr : prix moyen journalier

Chaque ligne est modélisée comme un document JSON dans MongoDB.

### 3. URL de base

<http://localhost:5000/api/v1>

Remplacer par l'URL de production en déploiement cloud.

### 3. Schéma des données (exemple de réservation)

```
{
  "hotel": "City Hotel",
  "arrival_date_month": "July",
  "adults": 2,
  "children": 1,
  "country": "USA",
  "booking_status": "Canceled",
  "adr": 120.50
}
```

Champ	Type	Obligatoire	Description
hotel	string	oui	Type d'hôtel
arrival_date_month	string	oui	Mois d'arrivée
adults	int	oui	Nombre d'adultes
children	int	oui	Nombre d'enfants
country	string	oui	Code pays ISO (FRA, PRT, etc.)
booking_status	string	oui	Canceled ou Not Canceled
adr	float	oui	Prix moyen journalier

## 5. Endpoints RESTful

### 5.1. POST /items

Ajouter une nouvelle réservation

- ✉ Corps requis : JSON (voir schéma ci-dessus)

- Réponse :

```
{
  "message": "Item ajouté",
  "id": "<ObjectId>"
}
```

- Erreur : Données invalides → 400

## 5.2. GET /items

### Lister toutes les réservations

- Réponse :

```
[
  {
    "_id": "...",
    "hotel": "City Hotel",
    "adults": 2,
    ...
  },
  ...
]
```

## 5.3. GET /items/<id>

### Récupérer une réservation par ID

- Exemple de requête :
- GET /api/v1/items/686a75c5f1090e20157ad30f**
- Réponse (200 - Succès) :

```
{
  "_id": "686a75c5f1090e20157ad30f",
  "hotel": "City Hotel",
  "arrival_date_month": "July",
  "adults": 2,
  "children": 1,
  "country": "USA",
  "booking_status": "Canceled",
  "adr": 120.50
```

```
}
```

- . Réponse (404 - Introuvable) :

```
{  
  "error": "Item non trouvé"  
}
```

- . Réponse (500 - Erreur serveur) :

```
{  
  "error": "Erreur interne du serveur"  
}
```

#### 5.4. GET /items/country/<country\_code>

Filtrer les réservations par pays

- Exemple de requête :

**GET /api/v1/items/country/USA**

- Réponse (200 - Succès) :

```
[  
 {  
   "_id": "...",  
   "hotel": "City Hotel",  
   "arrival_date_month": "August",  
   "adults": 2,  
   "children": 1,  
   "country": "USA",  
   "booking_status": "Not Canceled",  
   "adr": 98.75  
 }
```

```
},
```

```
...
```

```
]
```

. Réponse (404 - Aucun résultat) :

```
{
```

```
    "error": "Aucune réservation trouvée pour ce pays"
```

```
}
```

## 5.5. PUT /items/<id>

Modifier une réservation existante

- Exemple de corps JSON :

```
{
```

```
    "booking_status": "Not Canceled",
```

```
    "adr": 105.25
```

```
}
```

. Réponse (200 - Succès) :

```
{
```

```
    "message": "Item mis à jour"
```

```
}
```

. Réponse (404 - Introuvable) :

```
{
```

```
    "error": "Item non trouvé"
```

```
}
```

## 5.6. DELETE /items/<id>

Supprimer une réservation

- Succès :

```
{  
  "message": "Item supprimé"  
}
```

. Introuvable → 404

## 6. Codes de réponse http

Code	Signification
200	Succès
201	Création réussie
400	Requête invalide
404	Ressource non trouvée
500	Erreur interne serveur

## 7. Technologies utilisées

Composant	Technologie
Backend	Flask
Base de données	MongoDB (Atlas)
Client HTTP	Thunder Client
Validation	Python (validators.py)
Variables config	python-decouple
CORS	flask-cors

## Conclusion

Le projet **flask\_api** a permis de mettre en œuvre une **API RESTful robuste**, conçue pour l'analyse, la gestion et l'exploitation de **données réelles de réservations hôtelières** issues du jeu de données hotel\_bookings.csv disponible sur Kaggle. Grâce à l'intégration de technologies modernes telles que **Flask**, **MongoDB Atlas** et **Thunder Client**, le projet offre une plateforme flexible pour effectuer des opérations CRUD et générer des analyses statistiques à valeur ajoutée.

Les principaux objectifs du projet ont été atteints :

- Création d'une architecture API modulaire, extensible et maintenable.
- Mise en place d'une couche de validation solide pour sécuriser les données entrantes.
- Déploiement d'un backend documenté et structuré autour des besoins métiers (revenu mensuel, annulations, popularité par pays, etc.).
- Utilisation de pratiques de développement professionnel (logs, modularité, pagination, messages d'erreur standardisés).
- Tests rigoureux avec **Thunder Client** pour garantir la fiabilité de chaque route.

Au-delà de la simple manipulation de données, ce projet s'est inscrit dans une **démarche d'apprentissage avancé** des concepts clés du développement web backend, de la modélisation NoSQL, et de l'analyse de données au service d'un cas concret.