Restaurant Recommendation Service - Technical Documentation

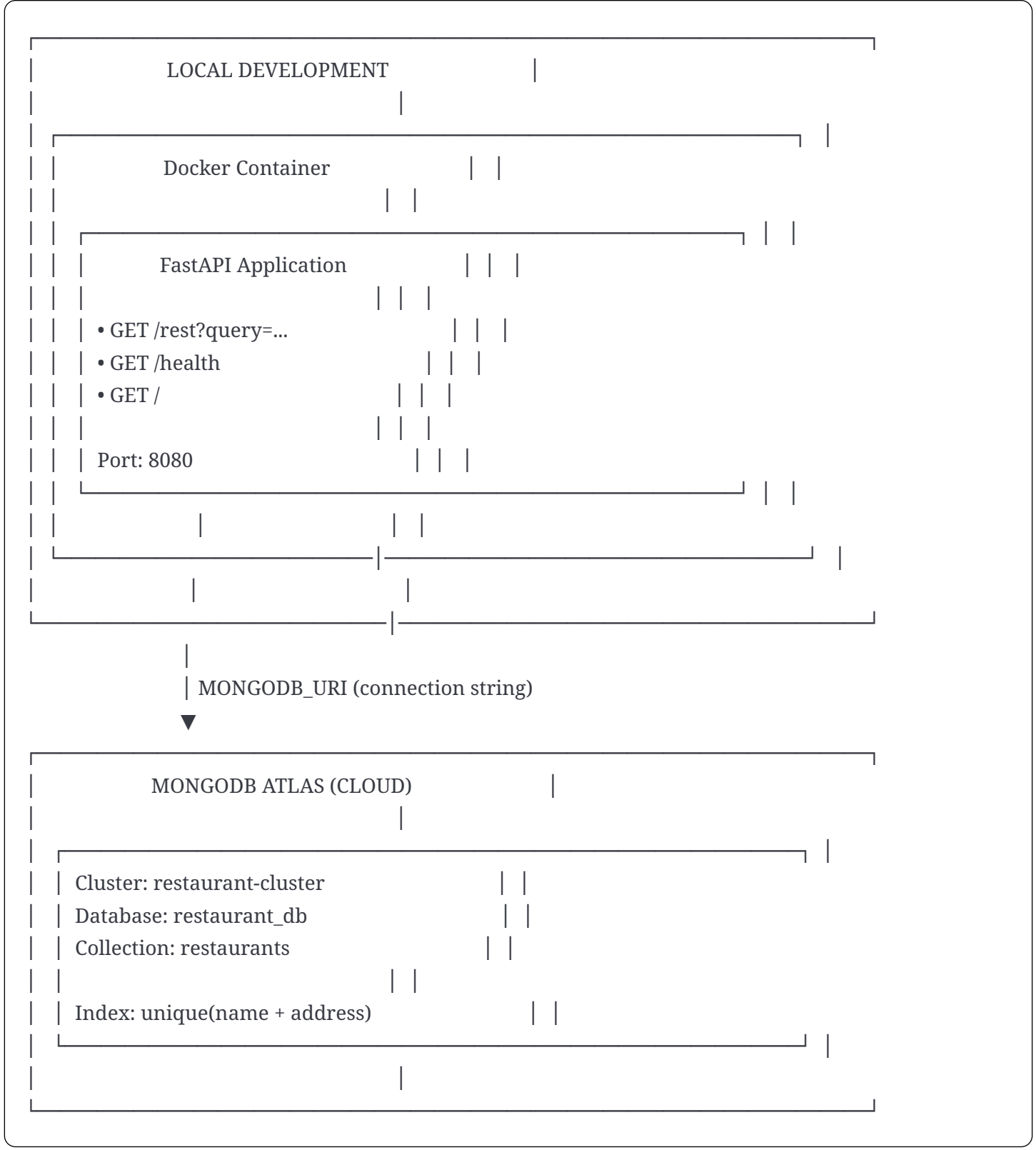
Project Overview

The Restaurant Recommendation Service is a RESTful API that provides restaurant recommendations based on free-text natural language queries. Built with Python/FastAPI and MongoDB Atlas.

Version: 1.0.0
Date: December 2024
Status: Application Complete, Ready for DevOps Phase

---

1. Architecture Diagram



---

2. Project Structure

```
restaurant-recommendation/
├── app/
│   ├── __init__.py          # Package marker
│   ├── config.py            # Environment variable management
│   ├── database.py          # MongoDB Atlas connection & operations
│   ├── main.py              # FastAPI application & endpoints
│   ├── models.py            # Pydantic data validation models
│   └── query_parser.py      # Free-text query parsing logic
├── scripts/
│   └── load_restaurants.py  # Data validation & loader script
├── restaurants/
│   └── restaurants.json     # Sample restaurant data (10 entries)
├── tests/
│   ├── test_api.py          # Model validation tests
│   └── test_query_parser.py # Query parser unit tests
├── .env.example             # Environment variable template
├── .dockerignore            # Docker build exclusions
├── docker-compose.yml       # Container orchestration
├── Dockerfile               # Application container definition
├── requirements.txt         # Python dependencies
└── README.md                # Quick start guide
```

---

3. File-by-File Documentation with Import Explanations

3.1 app/config.py

Purpose: Centralized configuration management via environment variables.

```python
import os
```

Import	Source	Why Used
`os`	Python Standard Library	Access environment variables via `os.getenv()` for configuration without hardcoding values

Key Configuration:

- `MONGODB_URI`: MongoDB Atlas connection string
- `DATABASE_NAME`: Target database name
- `PORT`: Server listening port (default: 8080)
- `LOG_LEVEL`: Logging verbosity (DEBUG, INFO, WARNING, ERROR)
- `LOG_FORMAT`: Log output format (json for production, text for development)

3.2 app/models.py

Purpose: Pydantic models for strict data validation (exactly 6 fields, no extras).

```python
from pydantic import BaseModel, Field, field_validator, ConfigDict
from typing import List, Union
import re
```

Import	Source	Why Used
`BaseModel`	pydantic	Base class for data validation models with automatic parsing
`Field`	pydantic	Define field constraints (min_length, description)
`field_validator`	pydantic	Create custom validation logic for specific fields
`ConfigDict`	pydantic	Configure model behavior (e.g., `extra="forbid"` to reject unknown fields)
`List`	typing	Type hint for lists (e.g., list of restaurants in response)
`Union`	typing	Type hint for multiple possible types (string OR list)
`re`	Python Standard Library	Regular expressions for time format validation (HH:MM, HHMM)

Key Design Decisions:

- `extra="forbid"`: Pydantic rejects any fields not in the schema (per spec: no extra fields)
- Time normalization: Accepts both HH:MM and HHMM, normalizes to HH:MM
- Vegetarian validation: Only accepts "yes" or "no", case-insensitive

---

3.3 app/database.py

Purpose: Async MongoDB operations using Motor driver for MongoDB Atlas.

```python
import logging
from typing import Optional, List, Dict, Any
from motor.motor_asyncio import AsyncIOMotorClient
from pymongo.errors import DuplicateKeyError
from app.config import config
```

Import	Source	Why Used
`logging`	Python Standard Library	Structured logging for connection status and errors
`Optional`	typing	Type hint for nullable values (e.g., `_client: Optional[...]`)
`List, Dict, Any`	typing	Type hints for return values (list of restaurants, query filters)
`AsyncIOMotorClient`	motor.motor_asyncio	**Async MongoDB driver** - non-blocking database operations for FastAPI
`DuplicateKeyError`	pymongo.errors	Catch duplicate key violations on unique index (name + address)
`config`	app.config	Access centralized configuration (MONGODB_URI, DATABASE_NAME)

Why Motor (not PyMongo)?

- Motor is the async version of PyMongo
- FastAPI is async-first; using sync PyMongo would block the event loop
- Motor wraps PyMongo with async/await support

Key Operations:

- `connect()`: Establish connection, create unique index
- `find_restaurants()`: Query with filter, exclude `_id` from results
- `insert_restaurant()`: Insert with duplicate detection

3.4 app/query_parser.py

Purpose: Parse free-text queries into MongoDB filter objects.

```python
import re
import logging
from datetime import datetime
from typing import Dict, Any, Optional, Tuple
from dataclasses import dataclass
```

Import	Source	Why Used
`re`	Python Standard Library	Regular expressions for extracting time patterns ("between 10:00 and 22:00")
`logging`	Python Standard Library	Log parsing decisions and errors
`datetime`	Python Standard Library	Get current server time for "open now" queries
`Dict, Any, Optional, Tuple`	typing	Type hints for function signatures
`dataclass`	dataclasses	Clean data structure for `ParsedQuery` (vegetarian, style, time constraints)

Parsing Flow:

1. Check for "vegetarian" keyword → yes/no
2. Scan for style keywords (italian, asian, steakhouse, mediterranean) → first match wins
3. Extract time constraints:
   - "between X and Y" → validate no midnight crossing
   - "opens at X" / "closes at X" → single constraint
   - No time → use current server time

Key Design Decision: Midnight Crossing Validation

- Per company clarification: queries like "between 22:00 and 02:00" return error
- Detected by: `start_time > end_time` after normalization

3.5 app/main.py

Purpose: FastAPI application with REST endpoints.

```python
import logging
import sys
from contextlib import asynccontextmanager
from typing import Optional
from fastapi import FastAPI, Query, Request
from fastapi.responses import JSONResponse
from app.config import config
from app.database import Database
from app.query_parser import process_query
from app.models import APIResponse
```

Import	Source	Why Used
`logging`	Python Standard Library	Application logging
`sys`	Python Standard Library	Direct logging output to stdout (container-friendly)
`asynccontextmanager`	contextlib	Manage FastAPI lifespan (startup/shutdown) for DB connections
`Optional`	typing	Type hint for optional query parameter
`FastAPI`	fastapi	Web framework class
`Query`	fastapi	Declare query parameters with validation/ documentation
`Request`	fastapi	Access request details for logging
`JSONResponse`	fastapi.responses	Return custom status codes (503 for unhealthy)
`config`	app.config	Access configuration
`Database`	app.database	MongoDB operations
`process_query`	app.query_parser	Parse free-text queries
`APIResponse`	app.models	Response model for OpenAPI documentation

Endpoints:

- `GET /rest?query=...` → Restaurant recommendations
- `GET /health` → Health check (database status)
- `GET /` → API information

Safe Logging:

- All logging wrapped in try/except
- Per spec: logging failures must never crash the application

---

3.6 scripts/load_restaurants.py

Purpose: Validate and load restaurants.json into MongoDB Atlas.

```python
import asyncio
import json
import logging
import sys
import os
from pathlib import Path
from typing import List, Dict, Any, Tuple
from pydantic import ValidationError
from app.models import Restaurant, REQUIRED_FIELDS
from app.database import Database
from app.config import config
```

Import	Source	Why Used
`asyncio`	Python Standard Library	Run async database operations from sync script
`json`	Python Standard Library	Parse restaurants.json file
`logging`	Python Standard Library	Log validation errors and progress
`sys`	Python Standard Library	Exit codes (0=success, 1=some invalid, 2=fatal)
`os`	Python Standard Library	Read RESTAURANTS_FILE environment variable
`Path`	pathlib	Cross-platform file path handling
`List, Dict, Any, Tuple`	typing	Type hints
`ValidationError`	pydantic	Catch Pydantic validation failures
`Restaurant, REQUIRED_FIELDS`	app.models	Validate against schema, check field names
`Database`	app.database	Insert validated restaurants
`config`	app.config	Access configuration

Behavior (per company clarification):

- Skip invalid entries (log errors, continue processing)
- Skip duplicates (same name + address)
- Exit code 1 if any invalid entries (for CI awareness)

3.7 requirements.txt

```
# Core dependencies
fastapi==0.109.0        # Web framework with async support
uvicorn[standard]==0.27.0  # ASGI server to run FastAPI
pydantic==2.5.3         # Data validation

# MongoDB (with TLS support for Atlas)
motor==3.3.2            # Async MongoDB driver
pymongo[srv]==4.6.1     # MongoDB driver (Motor dependency)
dnspython==2.4.2        # DNS resolution for mongodb+srv:// URIs

# Testing
pytest==7.4.4           # Test framework
pytest-asyncio==0.23.3  # Async test support
httpx==0.26.0           # HTTP client for API tests
mongomock==4.1.2        # Mock MongoDB for offline tests
```

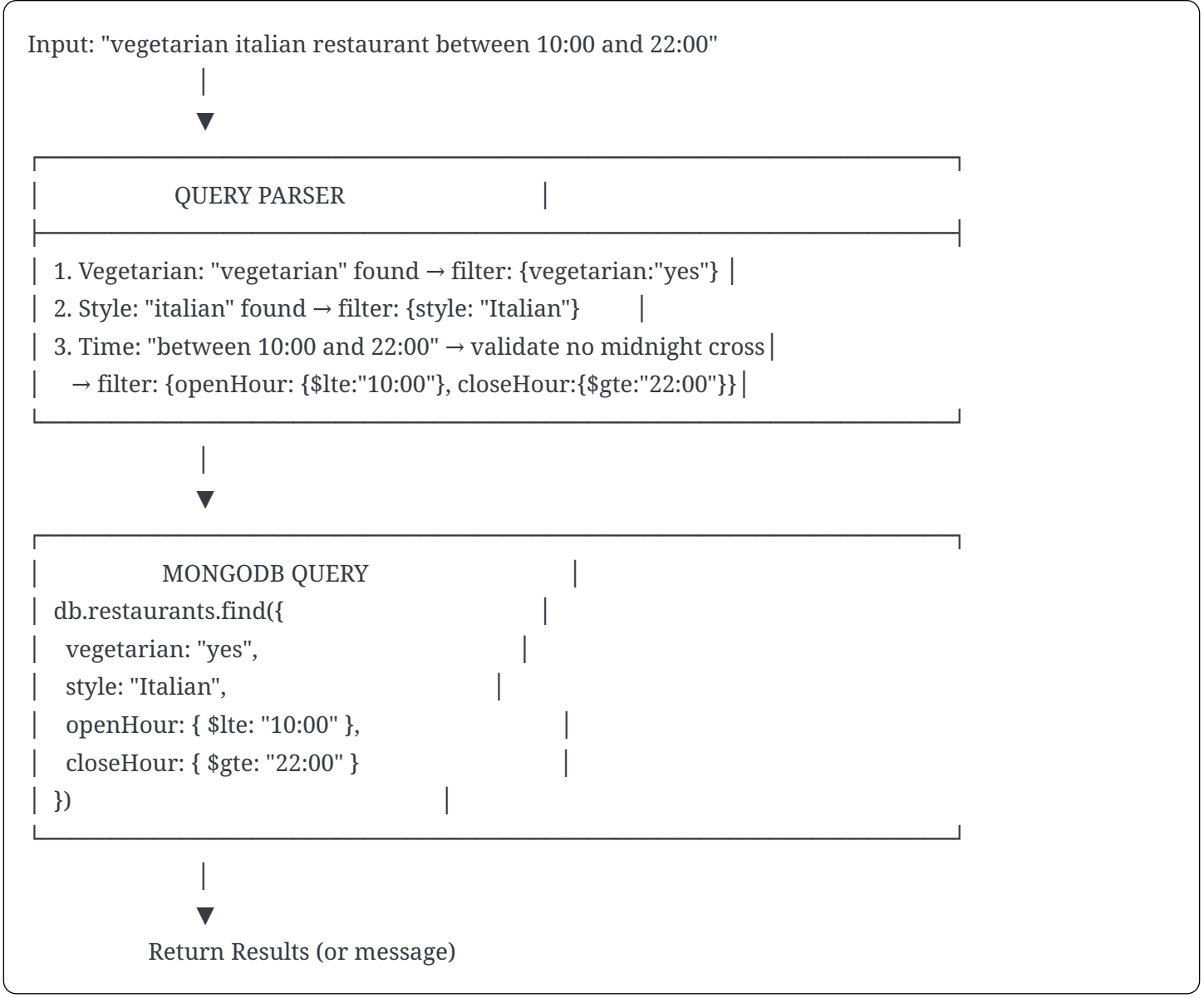
Why Each Dependency:

Package	Purpose
`fastapi`	Modern async web framework with automatic OpenAPI docs
`uvicorn[standard]`	Production-ready ASGI server (includes uvloop, httptools)
`pydantic`	Type-safe validation, integrates with FastAPI
`motor`	Async MongoDB driver for non-blocking operations
`pymongo[srv]`	Includes DNS support for Atlas connection strings
`dnspython`	Required for resolving mongodb+srv:// URIs
`pytest`	Standard Python test framework
`pytest-asyncio`	Run async tests with pytest
`httpx`	Async HTTP client for testing FastAPI
`mongomock`	In-memory MongoDB mock for unit tests without real DB

---

4. Business Logic Summary

Query Processing Flow



Confirmed Business Rules

Rule	Behavior	Source
No "vegetarian" in query	Return **only non-vegetarian**	Company confirmed
Multiple styles	**First match wins**	Company confirmed
No time specified	Use **server local time**	Company confirmed
Midnight crossing (22:00-02:00)	Return **validation error**	Company confirmed
Invalid JSON entries	**Skip and continue**	Default (not contradicted)
Pagination	**Not needed**	Company confirmed

---

5. Test Coverage

Unit Tests: 37 Passing

Category	Tests	Description
Time normalization	6	HH:MM, HHMM, single digit, invalid formats
Vegetarian extraction	2	Present → yes, absent → no
Style parsing	6	All 4 styles, no match, first match wins
Time constraints	7	Between, opens at, closes at, midnight crossing
Query parsing	4	Empty, valid, invalid, midnight crossing
Model validation	8	Valid, normalization, extra fields, missing fields
Required fields	4	Field detection, extras, missing

Running Tests

```bash
# All tests
pytest tests/ -v

# With coverage
pytest tests/ -v --cov=app
```

---

6. Environment Variables Reference

Variable	Required	Default	Description
`MONGODB_URI`	✅ Yes	-	MongoDB Atlas connection string
`DATABASE_NAME`	No	`restaurant_db`	Database name
`COLLECTION_NAME`	No	`restaurants`	Collection name
`HOST`	No	`0.0.0.0`	Server bind address
`PORT`	No	`8080`	Server port
`LOG_LEVEL`	No	`INFO`	DEBUG, INFO, WARNING, ERROR
`LOG_FORMAT`	No	`json`	`json` or `text`
`RESTAURANTS_FILE`	No	`/app/restaurants/restaurants.json`	Data file path

---

7. API Response Examples

Success with Results

json


```
{
  "restaurantRecommendation": [
    {
      "name": "Pasta Delight",
      "style": "Italian",
      "address": "Maskit St 35, Herzliya",
      "vegetarian": "yes",
      "openHour": "10:00",
      "closeHour": "22:00"
    }
  ]
}
```


No Results

json


```
{ "restaurantRecommendation": "There are no results." }
```


Empty Query

json


```
{ "restaurantRecommendation": "query is empty" }
```


Invalid Time

json


```
{ "restaurantRecommendation": "Invalid time format: 25:99" }
```


Midnight Crossing Error

json


```
{ "restaurantRecommendation": "Time ranges crossing midnight are not supported: 22:00 to 02:00" }
```