🍕 PizzaFlow: A Reactive Pizza Shop CRUD REST API

PizzaFlow is a modern and scalable RESTful API designed for a local pizza shop. Built with Spring Boot, Spring WebFlux, and MongoDB, the application showcases the power of reactive programming to deliver non-blocking, asynchronous operations. The API enables efficient management of pizzas and orders while ensuring high performance under load.

Features

Pizzas Create: Add new pizzas to the menu with attributes such as name, description, size, toppings, and price. Read: Fetch details of a specific pizza by ID or retrieve the entire pizza menu. Update: Modify existing pizza details. Delete: Remove pizzas from the menu.

Orders Create: Place an order for one or more pizzas. Read: Retrieve an order by ID or view all orders. Update: Update the status of an order (e.g., Preparing, Delivered). Delete: Cancel an order.

Why Use Reactive Programming?

This application utilizes Spring WebFlux and Reactive MongoDB to improve scalability and resource utilization by: Supporting non-blocking IO, allowing better throughput under high load. Utilizing Mono and Flux for efficient handling of asynchronous streams of data. Making the application more responsive and efficient compared to traditional blocking APIs.

Technologies Used Spring Boot: For building robust Java applications.

Spring WebFlux: To enable reactive programming and non-blocking operations.

MongoDB: A NoSQL database for data storage, integrated with reactive support.

Maven: Dependency and build management.

1. Design Diagram

The architecture for this project would follow a layered structure:
User Interface (Client)

↕

Controller Layer (REST APIs)

↕

Service Layer (Business Logic)

↕

Repository Layer (MongoDB Interactions via Spring Data Reactive MongoDB)

↕

Database (MongoDB)

Entity Relationships:

- **Pizza**: Contains name, description, listOfToppings, sizeOptions, price, etc.
- **Order**: Contains orderId, listOfPizzas (referencing Pizza), orderStatus, timestamp, etc.

2. Documentation

Project Overview

This project implements a Pizza Shop REST API to manage pizzas and customer orders. The API leverages **Spring Boot** for core functionality and **MongoDB** for database management. For scalability, we use **Spring WebFlux** for reactive, non-blocking operations.

Endpoints

Pizza Endpoints:

- **POST /pizzas** - Add a new pizza to the menu.
- **GET /pizzas** - Retrieve all pizzas or a specific pizza (by ID).
- **PUT /pizzas/{id}** - Update existing pizza details.
- **DELETE /pizzas/{id}** - Remove a pizza from the menu.

Order Endpoints:

- **POST /orders** - Place a new order.
- **GET /orders** - Retrieve all orders or a specific order (by ID).
- **PUT /orders/{id}** - Update the status of an order.
- **DELETE /orders/{id}** - Cancel an order.

Tech Stack

- **Backend Framework**: Spring Boot (with Spring WebFlux for reactive programming)
- **Database**: MongoDB (using Spring Data Reactive MongoDB)
- **Programming Language**: Java
- **Tools**: Maven, GitHub, Docker (for containerization, if needed)

3. Error Handling & Validation

Error Handling

To handle errors effectively:

1. **Global Exception Handling**: Use Spring Boot's @ControllerAdvice and @ExceptionHandler to define a centralized error-handling mechanism for the API.
2. **Custom Exceptions**: Create custom exception classes for specific scenarios like PizzaNotFoundException, OrderNotFoundException, etc.
3. **Standardized Response Structure**: Return consistent error response structures. For example:

{

"timestamp": "2025-03-30T10:57:00",

"message": "Pizza not found",

"details": "/pizzas/123"

}

4. **Reactive Error Handling**: Use onErrorResume in WebFlux pipelines to gracefully handle errors during data processing.

Validation Mechanisms

To ensure data integrity:

1. **Bean Validation**: Use annotations like @NotNull, @Size, @Pattern, etc., in entity classes. For example:

   @NotNull

   private String name;

   @Size(min = 1)

   private List<String> toppings;

   2. **Request Validation**: Validate incoming JSON payloads using @Valid in controller methods, combined with BindingResult to capture validation errors.

   3. **Custom Validators**: For complex validations, implement ConstraintValidator and annotate fields with your custom validation logic.

   4. **Database Constraints**: Leverage MongoDB schema design to enforce unique constraints or data type validations.