RESTful API Implementation Overview

Solution Overview

This document outlines the design and rationale behind the implementation of a RESTful API for managing entities. It includes the creation of necessary models, a mock repository for database interaction, and the implementation of CRUD (Create, Read, Update, Delete) endpoints within a controller. Each aspect of the solution is explained below.

Models

IEntity Interface

Purpose: Defines the structure of an entity, ensuring consistency across different entity types.
Approach: Using an interface allows for polymorphism and abstraction, enabling different entity types to implement common properties.

Entity Class

Purpose: Represents a concrete implementation of an entity.
Approach: Inherits from IEntity, ensuring adherence to the defined structure. Properties include addresses, dates, gender, deceased status, and an identifier.
Address, Date, Name Classes
Purpose: Define the components of an entity (address, date, name).
Approach: Encapsulate relevant information for each component, promoting code organization and clarity.

Repository

EntityRepository Class

Purpose: Simulates the data access layer for interacting with entities.
Approach: Uses an in-memory collection to store entities for simplicity. Provides methods for CRUD operations and utilizes LINQ queries for data retrieval and manipulation.

Controller

EntitiesController Class

Purpose: Handles HTTP requests related to entities.
Approach: Utilizes dependency injection to inject an instance of EntityRepository. Implements endpoints for listing entities, retrieving a single entity, creating, updating, and deleting entities. Incorporates parameter binding for query parameters and route parameters.

CRUD Endpoints

1.GET /api/Entities

Purpose: Retrieves a list of entities with optional filtering.
Approach: Utilizes LINQ queries to filter entities based on search terms, gender, date range, and countries. Returns filtered entities as an HTTP response.

Implementing Searching and Filtering in GET /api/Entities Endpoint

The GET /api/Entities endpoint retrieves a list of entities, optionally allowing searching and filtering based on various criteria. Here's how searching and filtering are implemented:

1. Searching

Purpose: Allows users to search for entities based on specific keywords.
Implementation:
Utilizes the search query parameter to specify the search term.

Searches within entity addresses and names for occurrences of the search term.
If the search term is found in any part of the address (e.g., country or address line) or name (e.g., first name, middle name, or surname), the entity is considered a match.
Uses LINQ queries to filter entities based on the search criteria.
2. Filtering
Purpose: Allows users to filter entities based on specific criteria such as gender, date range, and countries.
Implementation:
Utilizes query parameters (e.g., gender, startDate, endDate, countries) to specify filtering criteria.
Filters entities based on the provided criteria using conditional statements.
Supports filtering by gender, date range (birthdates), and countries.
Explanation
Flexibility: The implementation allows users to search and filter entities based on multiple criteria, providing flexibility in retrieving specific subsets of data.
User Experience: Users can easily narrow down their search results by specifying search terms or applying filters based on gender, date range, or countries.
Efficiency: Utilizes LINQ queries to efficiently filter entities within the in-memory collection, ensuring optimal performance even with large datasets.
Integration: Seamlessly integrates searching and filtering capabilities into the existing endpoint, enhancing the functionality of the API without introducing complexity.
Response: Returns the filtered list of entities as an HTTP response, ensuring that users receive relevant data based on their search and filter criteria.

2.GET /api/Entities/{id}
Purpose: Retrieves a single entity by its identifier.
Approach: Retrieves the entity from the repository based on the provided id. Returns the entity if found, or returns a 404 Not Found status if not found.

3.POST /api/Entities
Purpose: Creates a new entity.
Approach: Adds the entity to the repository using the AddEntity method. Returns the newly created entity in the response body with a 201 Created status.

4.PUT /api/Entities/{id}
Purpose: Updates an existing entity.
Approach: Checks if the provided id matches the id of the entity to be updated. If matched, updates the entity using the UpdateEntity method. Returns a 204 No Content status.

5.DELETE /api/Entities/{id}
Purpose: Deletes an entity by its identifier.
Approach: Retrieves the entity from the repository based on the provided id. If found, deletes the entity using the DeleteEntity method. Returns a 204 No Content status.

Conclusion
This solution provides a structured and maintainable approach to building a RESTful API for managing entities. By adhering to best practices, such as separation of concerns, dependency injection, and code abstraction, the implementation ensures scalability, flexibility, and ease of maintenance. Additionally, the use of a mock repository allows for easy testing and development, with the possibility of replacing it with a real database implementation in a production environment.