Detailed Technical Documentation

Project: Dynamic Mapping System

Table of Contents

Overview

The Dynamic Mapping System is a .NET-based application that performs data mappings between internal DIRS21 models and partner-specific models, such as Google's data models. This flexible and extensible mapping system allows mapping models in both directions (DIRS21 to partner and vice versa) while following SOLID principles and utilizing design patterns for modularity and maintainability.

Architecture

The system is organized into the following main components:

1. **Controllers**: Exposes mapping functionalities via REST API endpoints.
2. **Models**: Contains both internal (DIRS21) and external (partner-specific) data models.
3. **Services**:
   1. **MapperRegistry**: Manages all mapping strategies, serving as a registry for different mappers.
   2. **MapHandler**: Handles the actual mapping operations by using strategies from the MapperRegistry.
   3. **Mappers**: Specific mapping classes (e.g., RoomToGoogleRoomMapper) that define how data is mapped between different model types.

Diagram

The high-level architecture can be visualized as follows:

```
Client -> API (MappingController) -> MapHandler -> MapperRegistry -> Mappers -> Mo
```

Key Components and Classes

1. MappingController

- Acts as the API layer for the mapping functionalities.
    - Receives client requests and invokes `MapHandler` to perform mappings.

Main Methods:

- `MapRoomToGoogleRoom` : Maps a DIRS21 `Room` object to a `GoogleRoom` .
- `MapGoogleRoomToRoom` : Maps a `GoogleRoom` object to a DIRS21 `Room` .

2. MapperRegistry

- A singleton class that maintains a dictionary of available mapping strategies.
- Each mapping strategy is registered with a unique key based on the source and target model types (e.g., `Room-GoogleRoom`).

Key Methods:

- `RegisterMapper(string sourceType, string targetType, IMapperStrategy strategy)` : Registers a mapper strategy for specific model types.
- `GetMapper(string sourceType, string targetType)` : Retrieves the correct mapper for a given source and target type.

3. MapHandler

- Acts as a mediator that fetches the correct mapping strategy from `MapperRegistry` and applies it to perform the mapping.
- Accepts source and target types as parameters to dynamically resolve the appropriate strategy.

Main Method:

- `Map(object source, string sourceType, string targetType)` : Uses the appropriate mapper strategy from `MapperRegistry` to map the source object to the target type.

4. Mapping Strategies (Mappers)

- Individual classes implementing `IMapperStrategy` that define the mapping logic for each model type.
- Example: `RoomToGoogleRoomMapper` maps a `Room` object to a `GoogleRoom` object.

Example: RoomToGoogleRoomMapper

```
public class RoomToGoogleRoomMapper : IMapperStrategy
{
    public object Map(object source)
    {
        var room = source as Room;
        if (room == null)
        {
            throw new InvalidMappingException("Invalid source object. Expected Roor
        }

        return new GoogleRoom
        {
            GoogleRoomId = room.RoomId,
            RoomCategory = room.RoomType,
            MaxOccupancy = room.Capacity
        };
    }
}
```

Design Patterns

1. Strategy Pattern

- Enables each mapping to have its own strategy by implementing the `IMapperStrategy` interface.
- Mappers like `RoomToGoogleRoomMapper` and `GoogleRoomToRoomMapper` can be dynamically switched based on the source and target types.

2. Singleton Pattern

- `MapperRegistry` is a singleton, ensuring that only one instance is used across the application for managing mappers.

3. Dependency Injection

- The DI setup in `Program.cs` provides a modular, testable, and maintainable way to manage dependencies, such as `MapperRegistry` and `MapHandler`.

Dependency Injection Setup

The DI setup in `Program.cs` registers all services used throughout the application.

Program.cs Configuration

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container
builder.Services.AddLogging();

// Register MapperRegistry as Singleton with pre-registered mappers
builder.Services.AddSingleton<MapperRegistry>(serviceProvider =>
{
    var logger = serviceProvider.GetRequiredService<ILogger<MapperRegistry>>();
    var registry = new MapperRegistry(logger);
    registry.RegisterMapper("Room", "GoogleRoom", new RoomToGoogleRoomMapper());
    registry.RegisterMapper("GoogleRoom", "Room", new GoogleRoomToRoomMapper());
    return registry;
});

// Register MapHandler as Scoped
builder.Services.AddScoped<MapHandler>();
builder.Services.AddControllers();
builder.Services.AddSwaggerGen();
```

Instructions for Extending the System

To add new mappings between additional DIRS21 and partner models:

Create New Model Classes:

- Add the new model class to either `DIRS21Models` or `PartnerModels`.

Implement a New Mapper Strategy:

- Create a new class that implements `IMapperStrategy`.
- Define the mapping logic in the `Map` method.

Register the Mapper in MapperRegistry:

- Add the new mapper to `MapperRegistry` in `Program.cs`.
- Use a unique key based on source and target types.

Test the New Mapping:

- Verify the new mapping via Swagger or API testing tools.

Assumptions and Limitations

Assumptions

- **Single Directional Mapping:** Each mapping is implemented for specific source-to-target transformations (e.g., Room to GoogleRoom).
- **In-Memory Transformation:** All mappings are synchronous and do not involve database operations or other I/O-bound tasks.

Limitations

- **Current Scope**: Supports only Room and Reservation mappings. Extending this requires additional mappers and model classes.
- **Limited to DIRS21 and Google Models**: The existing setup focuses on these two model domains, though it can be extended.