Mapping System Architecture Documentation

Overview

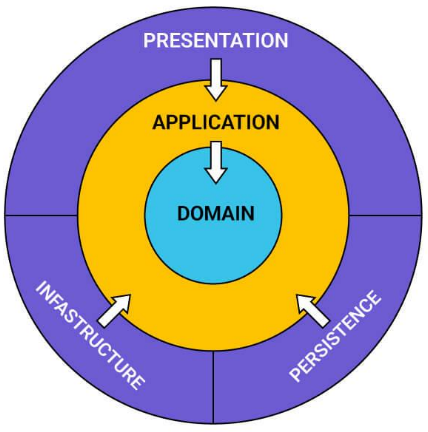
The Mapping System is designed to dynamically map data from a source object to a target object based on configurations defined in a JSON file. This system adheres to the **Clean Architecture** principles, separating concerns across distinct layers to ensure flexibility, scalability, and ease of maintenance. The architecture enables developers to add or modify mappings without changing the code, making the system extensible and suitable for integrating with third-party systems.

Project Requirements Recap

The system meets the following requirements:

1. **Mapping System Core**: Implements a core MapHandler class that manages mapping between various source and target formats.
2. **Supported Data Formats**: Provides mappings from internal DIRS21 models (e.g., Reservation, Room) to external models and vice versa.
3. **Extensibility**: Designed for easy extension, allowing new source/target formats and mapping rules to be added with minimal code modification.
4. **Error Handling and Validation**: Includes robust error handling for invalid mappings and incompatible formats, with meaningful error messages.
5. **Documentation**: This document provides architecture, key classes, methods, and guidelines for extending the system.

Architecture Layers



The Mapping System is divided into four layers:

1. **Domain Layer**: Defines core models and interfaces.

2. **Application Layer**: Contains business logic related to mapping and utilizes dependency injection for flexibility.
3. **Infrastructure Layer**: Manages other application APIs and I/O operations, such as loading JSON configuration files, and provides implementations for core interfaces.
4. **Presentation Layer**: Acts as the entry point for the console application, handling configuration loading and initializing the mapping process.
5. **Persistence Layer**: this Layer is responsible for managing data storage, typically interacting with a database or external data source. Since this project does not involve a database or require persistent data storage, a Persistence Layer is unnecessary.

Domain Layer

The **Domain Layer** contains core entities and data models representing the structure of data in the system. This layer is kept independent from other layers to facilitate reuse and portability.

- **Classes and Interfaces**:
  - **MappingConfiguration**: Represents the configuration needed for mappings, including TypeDefinition and MappingDefinition for each mapping rule.
  - **TypeDefinition**: Defines a target data type, specifying its properties and types.
  - **MappingDefinition**: Specifies how fields in a source type map to fields in a target type, including any conversion rules.
  - **FieldMapping**: Details each field mapping, including target fields, conversion types, and formatting requirements.
  - **ReservationMdl**: Represents an internal model for a reservation, used as a source/target in mappings.
  - **GreaterThanZeroAttribute**: A custom attribute for validation to ensure that properties (e.g., Id) have values greater than zero.
- **Responsibilities**:
  - **Data Representation**: Defines the data structure and relationships, serving as the basis for mappings.
  - **Validation**: Enforces constraints within models to ensure data integrity.

Application Layer

The **Application Layer** contains the core business logic for mapping and data transformation. It includes services for dynamic mapping and type creation, ensuring that mappings are executed according to the configuration.

- **Classes and Interfaces**:
  - **IMapAlgorithm**: Defines the contract for mapping execution, allowing flexibility to implement various mapping algorithms.
  - **DynamicMapper**: Implements IMapAlgorithm and provides the main logic for mapping data between models based on configuration.
  - **DynamicTypeBuilder**: Generates target types dynamically at runtime based on configuration, enabling flexibility in handling unknown target types.
  - **IMapHandler**: Interface that manages high-level mapping operations, coordinating dependencies and ensuring mapping flows smoothly.

    - **MapHandler**: Orchestrates the entire mapping process, interacting with DynamicMapper and loading configurations as needed.
    - **IJsonRepo**: This interface defines methods for interacting with JSON configuration files, enabling the Application layer to work with JSON data without depending on specific file-handling implementations. Since classes implementing this interface reside in the Infrastructure layer (which depends on the Application layer), IJsonRepo is located in the Application layer to avoid circular dependencies. Dependency injection is used to inject the Infrastructure implementation of IJsonRepo into Application layer components as needed.
- **Responsibilities**:
    - **Mapping Execution**: Manages the core logic for mapping source models to target models using configurations.
    - **Type Creation**: Builds new types dynamically based on definitions provided in the configuration, allowing for adaptable target models.
    - **Configuration Management**: Loads and manages mapping configurations, ensuring a data-driven approach to mapping.

Infrastructure Layer

The **Infrastructure Layer** handles interactions with external resources, including loading and managing JSON configuration files. It abstracts the data access, allowing the Application layer to remain independent of specific file-handling details.

- **Classes and Interfaces**:
    - **JsonRepo**: Implements IJsonRepo and loads mapping configurations from JSON, converting it into MappingConfiguration objects.
- **Responsibilities**:
    - **Data Access**: Provides a means to load and parse JSON configuration files, enabling dynamic control over mappings.
    - **Error Handling and Logging**: Manages file-related exceptions and ensures graceful handling of missing or malformed configurations.

Presentation Layer

The **Presentation Layer** serves as the entry point to the application, setting up dependency injection, initializing logging, and starting the mapping process.

- **Program.cs**:
    - Initializes configurations and dependencies using dependency injection.
    - Creates and invokes the MapHandler to perform mappings.
    - Manages application flow, logging, and orchestration of the mapping system.
- **Responsibilities**:
    - **Initialization**: Sets up logging, configuration paths, and dependencies.
    - **Mapping Orchestration**: Coordinates the entire mapping process by initializing MapHandler and triggering mappings.

Key Components

1. MappingConfiguration

Defines the structure of mappings using two main properties: `Types` and `Mappings`.

- **Types**: List of target types and their properties, used to create dynamic types.
- **Mappings**: List of mappings that define how fields in the source type map to fields in the target type, including any conversion details.

2. DynamicMapper

The `DynamicMapper` class performs mapping between source and target objects based on the configuration.

- **Responsibilities**:
  - **Field Mapping**: Maps fields from source objects to target objects according to `MappingConfiguration`.
  - **Type Conversion**: Applies necessary conversions, handling mismatched types and formatting.
  - **Error Handling**: Manages errors for missing fields, type mismatches, and invalid conversions.

3. DynamicTypeBuilder

Handles dynamic creation of target types based on `TypeDefinition` in the configuration, allowing for flexible target structures.

- **Responsibilities**:
  - **Type Creation**: Builds new types dynamically using reflection and `TypeDefinition`.
  - **Property Assignment**: Adds properties to dynamically created types, supporting flexible data structures.

4. MapHandler

The `MapHandler` class orchestrates the mapping process by interacting with `DynamicMapper` and `JsonRepo`.

- **Responsibilities**:
  - **Configuration Loading**: Loads mapping configurations from `JsonRepo`.
  - **Mapping Execution**: Uses `DynamicMapper` to apply mappings based on configuration, handling errors gracefully.

5. JsonRepo

Implements `IJsonRepo` to manage loading of JSON configuration files.

- **Responsibilities**:

- **Configuration Loading**: Reads JSON files and parses them into MappingConfiguration.
- **Error Handling**: Handles file-related exceptions, logging errors when configurations are missing or malformed.

Workflow

The mapping system follows this workflow:

1. **Configuration Loading**: JsonRepo loads the MappingConfiguration from JSON files, containing all the mapping rules and type definitions.
2. **Type Creation**: DynamicTypeBuilder creates target types dynamically if needed, using definitions from MappingConfiguration.
3. **Mapping Execution**: MapHandler uses DynamicMapper to map fields from the source to the target, applying conversions as specified.
4. **Error Handling**: Logs and handles any exceptions, such as type mismatches, missing properties, or conversion errors.

Extensibility and Customization

The system is designed for extensibility:

1. **Adding New Mappings**: Extend the JSON configuration with additional source/target mappings and conversion rules.
2. **Custom Conversion Logic**: Extend DynamicMapper to handle new data conversion types as needed.

Example JSON Configuration (mapping-config.json)

```
{
  "Types": [
    {
      "Name": "GoogleReservation",
      "Properties": {
        "GoogleReservationId": "string",
        "ReservationDate": "DateTime",
        "GuestName": "string"
      }
    }
  ],
  "Mappings": [
    {
      "SourceType": "DIRS21.Reservation",
      "TargetType": "GoogleReservation",
      "GenerateClassFlag": true,
      "Fields": {
        "Id": { "TargetField": "GoogleReservationId", "ConversionType": "IntToString" },
        "Date": { "TargetField": "ReservationDate", "ConversionType": "DateTimeToString", "Format": "yyyy-
MM-dd" },
        "CustomerName": { "TargetField": "GuestName" }
      }
    }
  ]
}
```

Error Handling

- **Type Mismatches**: DynamicMapper checks for mismatches between source and target types, logging errors and throwing exceptions if necessary.
- **Missing Properties**: Logs warnings for any missing source or target properties during mapping.
- **Configuration Issues**: JsonRepo handles missing or malformed configuration files, logging errors and failing gracefully.

Conclusion

This Mapping System provides a modular, extensible framework for dynamic data mapping between DIRS21 models and external partner data models. By following Clean Architecture principles, the system maintains high flexibility and supports easy customization, enabling new mappings and conversions to be added with minimal code changes.