DIRS21 - Technical Assessment

**Dynamic Mapping System**

Table of Contents

[1. Introduction 2](#_Toc179782585)

[2. Getting Started 2](#_Toc179782586)

[3. API Endpoints 3](#_Toc179782587)

[4. Key Classes and Methods 4](#_Toc179782588)

[5. Data Models 5](#_Toc179782589)

[6. Extending the API 5](#_Toc179782590)

[7. Deployment 6](#_Toc179782591)

[8. Testing 6](#_Toc179782592)

[9. Changelog 6](#_Toc179782593)

# 1. Introduction

Overview: The provided solution is a RESTful API that represent an extensible dynamic mapping engine/system using .Net 8  
The API is capable of mapping data between our internal DIRS21 C# data models and external data models used by our partners.  
It handles the conversion of our internal DIRS21 models to partner-specific formats and the mapping of incoming partner data to our internal DIRS21 models.

Architecture: This API consists of four layers.  
- The Data Access Layer (DAL – Class Library)  
- The Business Logic Layer (BLL – Class Library)  
- The Data Models Layer (DataModels – Class Library)  
- The RESTful API project (DynamicMapping – Web API)  
This four-tier architecture had the advantage of high reuse, strong scalability, low coupling and high efficiency.  
Design Patterns used: Repository, Generic Repository, Unit Of Work, Result Pattern   
  
Based on the requirements, it is not requested to implement an authentication/authorization module.

# 2. Getting Started

Prerequisites:   
This API can be run through Visual Studio 2022.  
This API can be tested without the need to be connected to a database however it is equipped to migrate its basic entities into a database on MS SQL Server.  
This API uses Swagger UI tool, therefore it can be consumed or tested directly through the Swagger Interface on the browser upon running it.  
It can also be consumed through Postman by sending a POST request to one of its endpoints  
Example Requests:   
1 - [/api/Room/Send](https://localhost:7112/api/Room/SendRoom)  
body =>   
{  
    "Id": 1,  
    "TargetType" : "Google",  
    "TargetTypeModel" : ""  
}

2 - [/api/Room/Receive](https://localhost:7112/api/Room/SendRoom)  
body =>   
{

    "SourceType": "Google",

    "SourceTypeModel": "",

    "SourceModel":

    {

        "Name": "The\_Main\_Room",

        "Description": "This is the main room",

        "Area": 100,

        "Floor": "5"

    }

}

# 3. API Endpoints

Overview: All the API endpoints includes an input class as parameters and output class as a response.  
All the output class always inherits from a “ReturnStatusModel” class that includes a “ReturnStatus” class and that class includes two properties Status Code and Return Message.  
  
**Detailed List of Endpoints**:  
  
1 – Send Room Data to Partner:   
 - Endpoint: HTTP Method (POST).  
 - URL: [/api/Room/Send](https://localhost:7112/api/Room/SendRoom)  
 - Description: this endpoint is consumed to send Room Data from our internal database to a specific partner.  
 - Parameters:   
\*input => SendRoomInput class   
{  
“Id”: long,  
“TargetType”: string,  
“TargetTypeModel”: string  
}

\*output => SendRoomOutput  
{  
“TargetModel”: Object  
}

2 – Receive Room Data from Partner:

- Endpoint: HTTP Method (POST).  
 - URL: [/api/Room/Receive](https://localhost:7112/api/Room/ReceiveRoom)  
 - Description: this endpoint is consumed to send Room Data from a partner-specific to our internal database.  
 - Parameters:   
\*input => ReceiveRoomInput class   
{  
“SourceType”: string,  
“TargetTypeModel”: string,  
“SourceModel”: Object  
}

\*output => ReceiveRoomOutput  
{  
“Succeeded”: Boolean  
}

3 – Send Reservation Data to Partner

4 – Receive Reservation Data from Partner  
  
- **Validation**:   
 Input Data Validation  
 Input Incompatible Data Format Validation against Source Data Model Format   
 Output Data Validation  
 Output Incompatible Data Format Validation against Target Data Model Format  
 Mapping Data Type Validation

- **Authentication**: No Authentication Required.

- **Errors**:   
 Global Error Handling (Custom Middleware).  
 Infrastructure Folder => ExceptionHandler Class

- **Logging**:  
 Logging Errors (Using Serilog Library & Implenting ILogger Interface)  
 Logs are written in txt file and saved within ExceptionLogs Folder  
 Logging Requests (Custom Middleware)  
 CustomMiddlewares Folder => RequestLoggingMiddleware Class

- **Caching**:  
 Caching Requests Responses (Caching Service)  
 Caching Folder in BLL => ICacheService Interface & ChacheService Class

# 4. Key Classes and Methods

**MapHandler**: Class in the BL layer, situated within the Core folder, that handles the mapping of all the transfers between the source and target data model objects.  
Methods: public static Object Map(Object data, string Source, string Target)  
This method receives as parameters:  
- data as Object   
- Source as string which includes the source data model name   
- Target as string which includes the target data model name.  
Based on both (source & target) data models name, it create an instance of these two classes.  
Then It gets their types and set them in variables.  
Then It fills the source data model instance with the data object sent.  
After that, it loops through the source type properties and find for each one a similar property having the same name, data type and write access grants within the target type properties in order to affect it with its data.  
  
If two properties, one from the source object and another one from the target object have similar names but different data types, then a descriptive error is thrown to the API consumer.

# 5. Data Models

Data Models class library is divided into two folders:  
1- Common: it contains the ResultStatusModel class which includes a property for the ReturnStatus Class and another property which is a list of ReturnStatus class as well as methods that implement the code and message error to return for validation and errors handling)  
2- Sections: this folder is divided into two other folders External and Internal.  
- External folder contains folders with the name of the partners and each partner folder contains a folder for each module which will be used within our system.  
In this last folder, a model class is created which structure is exactly as agreed with the partner about his data that will be shared with us.  
- Internal folder is divided into several folders, each one having a module name.  
In each module folder, there is a model that represents our internal entity model as well as a data transfer object model class located within a folder called “DTO”.   
This class contains all the actions input and output classes related to each module.

# 6. Extending the API

In order to add a new partner and activate dynamically the data transfer process between our internal models and its external models, here are the steps that should be taken by our developers:  
  
- **appsettings.json**: in the appsettings.json file, there is a section called “ExternalModels”.  
Within the json tree, under this node, a new node should be created with the name of the new partner like for example “Google”.  
Within the partner name node, a list of pairs (“Key”: “Value”) should be added where Key represents the module or the entity name which makes part of the data transfer objects list that will be shared between this customer and our system.  
Value represents the namespace path + the entity model name.  
example:  
"Google": {

"Room": "DataModels.Sections.External.Google.Room.RoomModel",

"Reservation": "DataModels.Sections.External.Google.Reservation.ReservationModel"

}

- **DataModels layer**: In the DataModels class library, within the Section»External folder, a new folder should be added having the name of the new partner like for example “Google”.  
Under “Google” folder, a folder for each entity (that will be included within the data transfer process) should be also added like for example “Room” and within that folder a model class should be created including all the properties needed to be shared between this partner and our system and they should be having the same name and datatype as our internal model.  
Those folders and classes structure and hierarchy as well as the model classes nomination should be always followed by our developers.

- **Controllers**: Within the controller related to each entity like for example “RoomController”, let’s say that this controller includes two main methods that deal with the data sharing process which are consecutively “SendRoomToPartner()” and “ReceiveRoomFromPartner()” methods.  
Within the “SendRoomToPartner()” method, a new case should be added under the switch statement, where the related model name will be pulled from the appsettings.json file and assign it to the input class “TargetTypeModel” property.  
Within the “ReceiveRoomFromPartner()” method, a new case should be added under the switch statement, where the related model name will be pulled from the appsettings.json file and assign it to the input class “SourceTypeModel” property.   
The input class “SourceModel” property content which is of type Object should be deserialized or unboxed into the related partner entity model type.

# 7. Deployment

# 8. Testing

# 9. Changelog