**Step 1: Register Controllers in Dependency Injection**

You can add the controllers to the DI container in Startup.cs (for ASP.NET Core 3.1/5) or Program.cs (for .NET 6/7/8) using AddTransient, AddScoped, or AddSingleton. These methods determine the lifecycle of the controllers.

**Example in .NET 6/7/8 (Program.cs)**

csharp

CopyEdit

builder.Services.AddTransient<BusesController>();

builder.Services.AddTransient<LabourDetailsController>();

**Example in .NET Core 3.1/5 (Startup.cs)**

csharp

CopyEdit

public void ConfigureServices(IServiceCollection services)

{

services.AddTransient<BusesController>();

services.AddTransient<LabourDetailsController>();

}

Here, we are telling the DI container to create a new instance of these controllers whenever they are requested.

**Step 2: Inject Controllers into RebuiltPartsController**

Now, you can inject the BusesController and LabourDetailsController into the constructor of RebuiltPartsController. This will allow you to call their methods directly.

**Updated RebuiltPartsController.cs**

csharp

CopyEdit

public class RebuiltPartsController : ControllerBase

{

private readonly IDbConnection \_dbConnection;

private readonly BusesController \_busesController;

private readonly LabourDetailsController \_labourDetailsController;

public RebuiltPartsController(

IDbConnection dbConnection,

BusesController busesController,

LabourDetailsController labourDetailsController)

{

\_dbConnection = dbConnection;

\_busesController = busesController;

\_labourDetailsController = labourDetailsController;

}

private async Task<IEnumerable<RebuiltPartsViewModel>> GetRebuiltPartsData()

{

try

{

// Base query for rebuilt parts data

var rebuiltPartsQuery = @"

SELECT rbm.RbMasterlistId,

rbm.[RebuiltStockNum],

rbm.[MmsStockCode],

rbm.[Keyword],

rbm.JobNumber,

rbm.[CorePartNum],

rbm.[DetailedDesc],

rbm.[CoreCharge],

rbm.[EstimatedCost],

rbm.[BuyNewCost],

rbm.[RemanCost],

rbm.[ExternalCost]

FROM [SBCES].[RbMasterlist] rbm";

var rebuiltParts = (await \_dbConnection.QueryAsync<RebuiltPartsViewModel>(rebuiltPartsQuery)).ToList();

foreach (var part in rebuiltParts)

{

// Call the BusesController method to get bus details

var busDetails = await \_busesController.GetBusDetails(part.RebuiltStockNum);

part.VehicleSeries = busDetails.Select(bus => new ListOfBusesModel

{

ListId = bus.ListId,

Description = bus.Description

}).ToList();

// Call the LabourDetailsController method to get labour details

var labourDetails = await \_labourDetailsController.GetLabourDetailsData(part.RebuiltStockNum);

part.LabourDetailsRebuiltParts = labourDetails.ToList();

}

return rebuiltParts;

}

catch (Exception ex)

{

// Handle errors gracefully

return null;

}

}

}

**Step 3: Adjust Controllers for Internal Use**

Since BusesController and LabourDetailsController are now used internally, you may want to make their methods more direct or remove attributes like [HttpGet] because they’re no longer strictly HTTP endpoints.

For example, update BusesController.cs like this:

**Updated BusesController.cs**

csharp

CopyEdit

public class BusesController : ControllerBase

{

private readonly IDbConnection \_dbConnection;

public BusesController(IDbConnection dbConnection)

{

\_dbConnection = dbConnection;

}

public async Task<IEnumerable<BusesModel>> GetBusDetails(string rebuiltStockNum)

{

var query = @"

SELECT rlb.[RebuiltStockNum],

lb.[ListId],

lb.[Description]

FROM [SBCES].[RbListOfBuses] rlb

INNER JOIN [SBCES].[ListOfBuses] lb ON rlb.[ListId] = lb.[ListId]

WHERE rlb.[RebuiltStockNum] = @RebuiltStockNum";

return await \_dbConnection.QueryAsync<BusesModel>(query, new { RebuiltStockNum = rebuiltStockNum });

}

}

Similarly, for LabourDetailsController.cs:

**Updated LabourDetailsController.cs**

csharp

CopyEdit

public class LabourDetailsController : ControllerBase

{

private readonly IDbConnection \_dbConnection;

public LabourDetailsController(IDbConnection dbConnection)

{

\_dbConnection = dbConnection;

}

public async Task<IEnumerable<LabourDetailsRebuiltPartsViewModel>> GetLabourDetailsData(string rebuiltPartNum)

{

var query = @"

SELECT EL.LINKNUMBER,

EL.LabourDefn AS [LabourType],

EL.CostCentre AS [CC],

EL.Task AS [Task],

REPLACE(EL.Usage, ',', '.') AS [Usage],

EL.HrsReqd AS [Time],

ES.RatePerHour AS [RatePerHour],

EL.TimeAddition AS [TimeAddition]

FROM [SBCES].[EmployeeLabour] EL

JOIN SBCES.EmplClass EC ON EL.LabourDefn = UPPER(EC.LabourDefn)

JOIN SBCES.EmployeeSalaries ES ON EC.WageGroup = ES.WageGroup

WHERE EL.[LinkNumber] = @RebuiltPartNum";

return await \_dbConnection.QueryAsync<LabourDetailsRebuiltPartsViewModel>(query, new { RebuiltPartNum = rebuiltPartNum });

}

}

**Summary of Changes**

1. **Registered Controllers in Startup.cs or Program.cs:**

csharp

CopyEdit

builder.Services.AddTransient<BusesController>();

builder.Services.AddTransient<LabourDetailsController>();

1. **Injected Controllers in RebuiltPartsController:**

csharp

CopyEdit

private readonly BusesController \_busesController;

private readonly LabourDetailsController \_labourDetailsController;

1. **Called Methods from Injected Controllers:**
   * Use \_busesController.GetBusDetails() to fetch bus details.
   * Use \_labourDetailsController.GetLabourDetailsData() to fetch labour details.
2. **Optional Cleanup:**
   * Removed [HttpGet] from BusesController and LabourDetailsController methods since they are now used as internal logic, not HTTP endpoints.

**Pros and Cons of This Approach**