**Optimized Code with Parallel Execution**

Here’s how you can refactor your GetRebuiltPartsData method to improve performance:

csharp

CopyEdit

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

{

try

{

// Step 1: Query base data for rebuilt parts

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();

if (!rebuiltParts.Any())

{

return rebuiltParts;

}

// Step 2: Prepare parallel tasks for fetching data

var tasks = rebuiltParts.Select(async part =>

{

// Fetch bus details in parallel

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

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

{

ListId = bus.ListId,

Description = bus.Description

}).ToList();

// Fetch labour details in parallel

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

part.LabourDetailsRebuiltParts = labourDetails.ToList();

return part;

});

// Step 3: Await all tasks concurrently

var completedParts = await Task.WhenAll(tasks);

return completedParts;

}

catch (Exception ex)

{

// Handle errors gracefully

return null;

}

}

**How It Works**

1. **Parallel Execution:**
   * Instead of processing GetBusDetails and GetLabourDetailsData for each part sequentially, the Select statement creates a list of tasks that will execute these calls in parallel for all parts.
   * Task.WhenAll waits for all the tasks to complete concurrently, drastically improving the speed for large datasets.
2. **No Blocking:**
   * The method remains asynchronous and non-blocking, ensuring that it scales well under high-load scenarios.

**Performance Considerations**

While this optimization improves performance, there are some considerations to keep in mind:

1. **Batch Processing:**
   * If the number of items in rebuiltParts is very large (e.g., thousands), sending too many parallel requests might overwhelm the database or APIs being called. In such cases, consider processing items in smaller batches. Here's an example of **batching**:

csharp

CopyEdit

const int batchSize = 50; // Process 50 items at a time

for (int i = 0; i < rebuiltParts.Count; i += batchSize)

{

var batch = rebuiltParts.Skip(i).Take(batchSize);

var tasks = batch.Select(async part =>

{

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

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

{

ListId = bus.ListId,

Description = bus.Description

}).ToList();

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

part.LabourDetailsRebuiltParts = labourDetails.ToList();

return part;

});

// Wait for the batch to complete

await Task.WhenAll(tasks);

}

1. **Database Connection Pooling:**
   * Ensure your database connection (\_dbConnection) supports connection pooling (most providers like SQL Server do by default). This avoids running out of connections when executing many concurrent requests.
2. **Controller Dependency Injection:**
   * If your \_busesController and \_labourDetailsController are injected as transient dependencies, they might also need to handle concurrent calls appropriately. Ensure they are stateless or thread-safe.
3. **Error Handling:**
   * If one of the parallel tasks fails (e.g., GetBusDetails for one specific part), the whole operation might throw an exception. You can add error handling for each individual task:

csharp

CopyEdit

var tasks = rebuiltParts.Select(async part =>

{

try

{

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

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

{

ListId = bus.ListId,

Description = bus.Description

}).ToList();

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

part.LabourDetailsRebuiltParts = labourDetails.ToList();

}

catch (Exception ex)

{

// Log the error and continue with the next part

Console.WriteLine($"Error processing part {part.RebuiltStockNum}: {ex.Message}");

}

return part;

});

var completedParts = await Task.WhenAll(tasks);

**Why This Is Faster**

Sequential execution (foreach loop) means the await calls block each subsequent iteration until the current one finishes. This results in **N sequential calls**, with a total execution time of:

text

CopyEdit

Execution time ≈ N \* (Time for GetBusDetails + Time for GetLabourDetailsData)

Using parallel execution with Task.WhenAll, the total execution time is closer to the time taken for the **slowest individual task**:

text

CopyEdit

Execution time ≈ Max(Time for GetBusDetails, Time for GetLabourDetailsData)

For example:

* If GetBusDetails takes 100ms and GetLabourDetailsData takes 200ms, and there are 50 parts, **sequential execution** takes ~15 seconds (50 \* (100 + 200) ms), while **parallel execution** takes ~200ms (plus overhead for task management).