Performance Analysis

Contents

[Improve Ability to Analyze 2](#_Toc204189428)

[General Approaches to Improve Performance 2](#_Toc204189429)

[Notes 3](#_Toc204189430)

[Caching Resources in the Browser 3](#_Toc204189431)

[Overview of Current Transaction Duration 6](#_Toc204189432)

[Reduce the Database Transactions 6](#_Toc204189433)

[Fix Synchronous Calls 8](#_Toc204189434)

[Review Methods With High p99 Transaction Duration 9](#_Toc204189435)

[OrderAppService.EditOrderLine 9](#_Toc204189436)

[EditJob 10](#_Toc204189437)

[EditOrder 10](#_Toc204189438)

[GetEditForOrderInternal 11](#_Toc204189439)

[GetOrders 12](#_Toc204189440)

[GetQuotes 12](#_Toc204189441)

[CopyOrder 13](#_Toc204189442)

[SetOrderLineIsComplete 14](#_Toc204189443)

[CancelDispatch 14](#_Toc204189444)

[SetNotificationAsRead 15](#_Toc204189445)

[SetIsVerifiedForTickets 15](#_Toc204189446)

[DeleteOrderLineTrucks 18](#_Toc204189447)

[UpdateOrderLineDates 19](#_Toc204189448)

[DeleteTrackableOrderEmails 20](#_Toc204189449)

[Emails/TrackEvents 21](#_Toc204189450)

[Reducing Database Transactions 21](#_Toc204189451)

[Long Running Queries 22](#_Toc204189452)

[Miscellaneous Notes 23](#_Toc204189453)

# Improve Ability to Analyze

We don’t currently know if our performance issues are due to Thread Pool Exhaustion, race conditions, Garbage collection, or something else. Today we aren’t any more clear about the issue than we were 3 weeks ago.

We recently added custom metrics for usedWorkerThreads, maxWorkerThreads, maxCompletionPortThreads, and AvailableThreads? We can view these in App Insights. Are there other metrics we need to track to determine the issue. These results suggest that ThreadPoolExhaustion is not the issue. Note how low the UsedWOrkerThreads is.

A screenshot of a computer

AI-generated content may be incorrect.

A screen shot of a graph

AI-generated content may be incorrect.

While writing these transactions may add a little overhead, I think we can afford that right now to identify the issue. I think this overhead is minimal when you consider the small resource utilization (of the resources we can see).

**Are there other metrics we need to add?**

# General Approaches to Improve Performance

Below are some general considerations for how we might improve performance. The goal is to try to determine the cause of the problem. Then we can approach the below in a manner that will give us the greatest chance of improvement per time spent.

For example, while there may be an issue with locks on resources at highr transaction volumes, if it is quicker to reduce the transaction volume rather than reengineering how we lock across the application, we might choose to reduce the transaction volume to buy us some time to do the larger effort fix.

1. Caching static resources on the front-end.
2. Don’t query data that is seldomly used until it is needed. Some items may be queried and loaded for every page view.
3. Reduce the database transactions.
4. Make sure async is used appropriately everywhere throughout the app and ABP components.
5. Ensure connection pools set appropriately.
6. Evaluate transaction necessity – for readonly operations, consider if transactions are needed.
7. Reduce the amount of data or width of data returned.
8. Reduce the time database transactions are held open
   1. Reduce multiple database calls in a loop.
   2. Simplify the logic to not require as many queries in the same transaction.
   3. Consider where sprocs might make sense.
   4. Consider what isn’t critical to be in a transaction and do this operation outside the database transaction.
9. Consider using batch operations and consolidation operations into fewer transactions.
10. Backend data caching.
11. Front-end data caching.
12. Review the locking of resources.

# Notes – things to check

DbContext and HttpContext are not thread safe. Do not try to run parallel tasks that use the same context.

Services that depend on IHttpContextAccessor, userManager, or similar, should not be singletons.

Scoped and Transient services should not be injected into a Singleton. If a singleton service (e.g., registered as AddSingleton) depends on a scoped (AddScoped) or transient (AddTransient) service, it can cause runtime errors or unexpected behavior.

Scoped services are meant to live within a single HTTP request. Using them outside (e.g., in background threads, static contexts, or singletons) is improper.

Ensure you do not store DbContext or repositories in static fields or singletons.

# If you use background jobs, use **IServiceScopeFactory** to create a scope for resolving scoped services.

# Caching Resources in the Browser

While most of these won’t impact the long delays we have been seeing due to suspected threadpool exhaustion, they will can significantly cut page load time after the initial page load.

Below is a waterfall of the resources loaded for a minimal page. This is the “Redirection Page”. Do we need to dynamically generate the GetAll and the GetScripts every time we come to a page? Could these be cached for every visit to a specific page, or is there a reason these have to be dynamic other than that it was built this way to prevent us from skipping the AspNetZero license check?

None of the two calls were related to the license check.

1. AbpServiceProxies/GetAll lists the JS proxies for the public API methods and it’s the same for all users and never changes for a given app version (except for local builds that have the same version number). We had plans to make it more dynamic, but until that is done it definitely can and should be cached. On the backend side, it is generated once, on the first request, and then stored in a static variable.
2. AbpScripts/GetScripts has a mix of a lot of static and some dynamic data. It lists:

* User session, which means at the very least it has to be cached per each individual user. That session also includes organization units assigned to the user, office name, office id, customer name, user email, and whether the user has access to all offices. So, if it is cached on the frontend, we’d need to make sure we can somehow invalidate the frontend cache on all of the possible session related changes. E.g. the addition of the new office would mean they no longer have access to all offices; Customer rename will mean the customer users must receive a different customer name; different user login obviously had to get a different session, even if logged out by a timeout and not intentionally; any ways to assign or remove an Organization Unit to a user has to invalidate the cache too.
* List of languages and the current language for the user. Right now we have a hardcoded once-per-launch list of languages so it’s not a concern, but if proceeded with the cache, later we will need a way to invalidate the frontend cache if a list of languages or the current language for the user/tenant/application changes.
* Localization sources and their entire contents – does not change (anymore) within the same app build as it is hardcoded in an XML. Not long ago it was modifiable through the database so the placement of those in this dynamic list made sense back then, but not anymore. This accounts for lines 50 through 2282 out of 3698, and these are a lot longer than the other lines so in size it’s probably most of the file.
* All permissions granted to the specific user (or their role, or default, etc – the combined result).
* Navigation menu. Doesn’t seem to be used and I think can safely be removed, at least for now. (lines 2788 – 3403, i.e. also a lot of lines)
* Some settings and their values – the ones marked to be sent to the frontend. Some probably can have a mixed application/tenant/user scope. So changing a single tenant settings must be able to somehow invalidate the cache of all affected users.

The caching itself is always easy, but all of the above makes the invalidation very difficult. For one, we do not have a way to get a historical diff between the setting state at a given date and now to know if it needs to be invalidated. The goal of saving the time does not feel achievable for this script, but we can save some bandwidth by calculating it twice – once on each cshtml render that includes the script, and once on the actual request for data. Then, we can append md5 hash of the script generated on the first step on the backend, append it like “/GetAll?md5=abcd...” on the view and mark the URL to be cached on the frontend. Then they should download less data after the very first request until the contents and the hash changes for any reason on the next page load.

We can try to move localization strings from GetScripts to AbpServiceProxies/GetAll despite the unmatching name (or add a third method to combine the static AbpServiceProxies/GetAll and static localization data) to utilize the stronger caching for this more static data.

Based on Aleksey’s comments above, I think the only things remaining on this topic is to cache the ABPServiceProxies/GetAll and remove the unnecessary parts of the ABPScripts/GetScripts. Later if we decide to add localization, we can figure out where to best put that code.

A screenshot of a computer

AI-generated content may be incorrect.

Can we cache the results of the 4 calls at the bottom in IndexDb to cut the load time significantly? Or for GetUserNotifications, do we instead get a count of unread notifications to show the number of unread notifications, but not load the notifications until they click the link? There is 1.5 seconds that could be reduced from the above page load across all those resources.

These same resources are loaded for almost every page in the application and shouldn’t need to be retrieved from the server every page load. Changing this will result in fewer database calls, in addition to the faster load times.

# Overview of Current Transaction Duration

Below is a graph of transaction duration for prod2. You can see how the median transaction duration in general increases with request volume. Note that over the past week, while the transactions are slightly increased, the transaction duration has trended down with the changes made. The async changes decreased the median duration from 88.6 to 65.3.

A screenshot of a graph

AI-generated content may be incorrect.

Below is the p99 graph of transaction duration and request count. The transaction duration doesn’t follow the request volume directly and is being impacted by some other variable. The changes this week have improved the situation, but have not corrected the underlying issue. On Friday I upgraded the prod2 app service to a P3V3 with 3 instances at the peak times to improve the performance. That is why there is an improvement Friday. While it helped, it didn’t resolve the issue.

This behavior suggest thread pool starvation. However, we need to add custom instrumentation to verify this.

A screenshot of a graph

AI-generated content may be incorrect.

# Reduce the Database Transactions

The database transactions can be reduced by some of the following methods:

* Reduce the amount of synching between the RN Driver App and the API.
* Determine why we have so many of the high volume transactions. Is there a way to reduce them?
* Add a cache for the sets of data that are frequently retrieved. Especially if these sets of data are long lived.

Below is the Transaction Duration sorted by number of transactions descending. It can be used as the basis for prioritizing what to cache and what to reduce.

A screenshot of a computer

AI-generated content may be incorrect.

The graph below shows the transaction duration and distribution for Account/Login. Note that it is involved in a lot of slow transactions. Is this just because of its frequency and it just happens to be running at the same time as the other transactions causing the issue, or is it the cause?

The image below brings up several questions.

* Why do we have 11k/day average logins? That seems high considering the number of users (including trucks). If the tokens are working correctly, each user wouldn’t need to login more than 10 times per day would they?
* Why do we have so many slow login transactions on the weekend when hardly anyone is using the system?

A screenshot of a computer

AI-generated content may be incorrect.

Isolating the graph to Sunday 6/15 looks like the below image. Note how the volume of logins is only slightly lower than the weekly average although the volume of other requests is significantly reduced.

A screenshot of a computer

AI-generated content may be incorrect.

# Fix Synchronous Calls

Finish the changes to make data access calls async. This includes the login call. What portion of async calls still exist after we deploy 2.9.9.7 (same as 2.9.9.6).

See task #15555 for a list of some opportunities. It may include some that have already been fixed.

After fixing the known synchronous calls and deploying 2.9.9.6 and .9, we still have some slower than expected transactions. Should they be profiled to see if they have any synchronous calls?

# Review Methods With High p99 Transaction Duration

Things to think about when evaluating each of the below methods which had large delays today. This will be the list of items in the above section.

* Can we simplify or remove some of the logic?
* Are we doing something in a loop that could be brought back in a set?
* If we are looping, do we need to loop through while holding a single UOW open, or can we loop through with each item being updated in separate UOWs?
* Are we using async appropriately.
* Do we prevent double clicking on the save button for any views with a save button? Do we prevent clicking the button a second time before the dialog or form is closed or the first action has completed?

The screenshot shown below shows various performance statistics. I think the ones highlighted in red are the ones to evaluate first because they have a bad 99p and run frequently. The ones in yellow should be evaluated next.



## OrderAppService.EditOrderLine

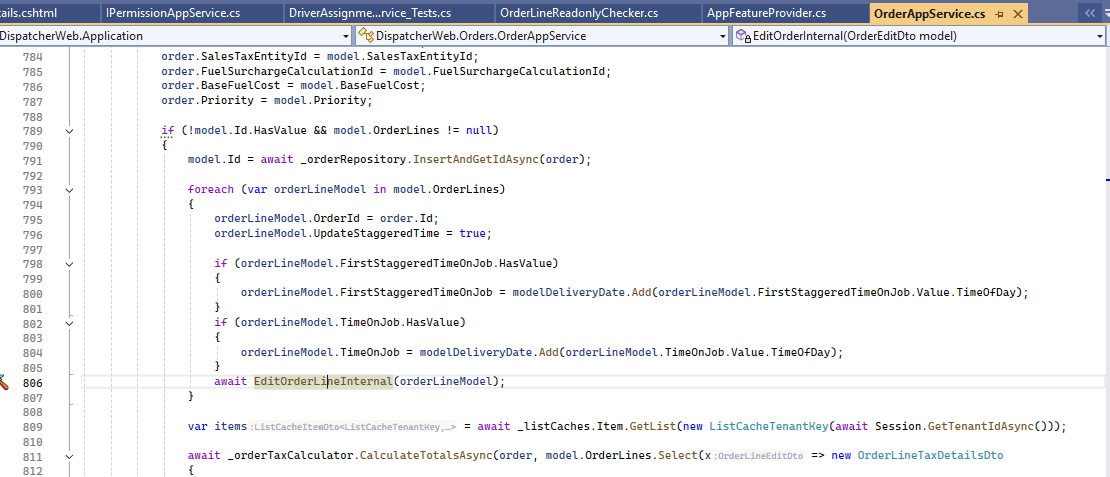
Should we check the permissions up front instead of in the middle of the transaction? It looks like the controller that calls the EditOrderLinesInternal already checks permissions. If they don’t have the appropriate permissions, does it make sense to do any of what is done in EditOrderLineInternal or should it return before doing wasted work?

## EditJob

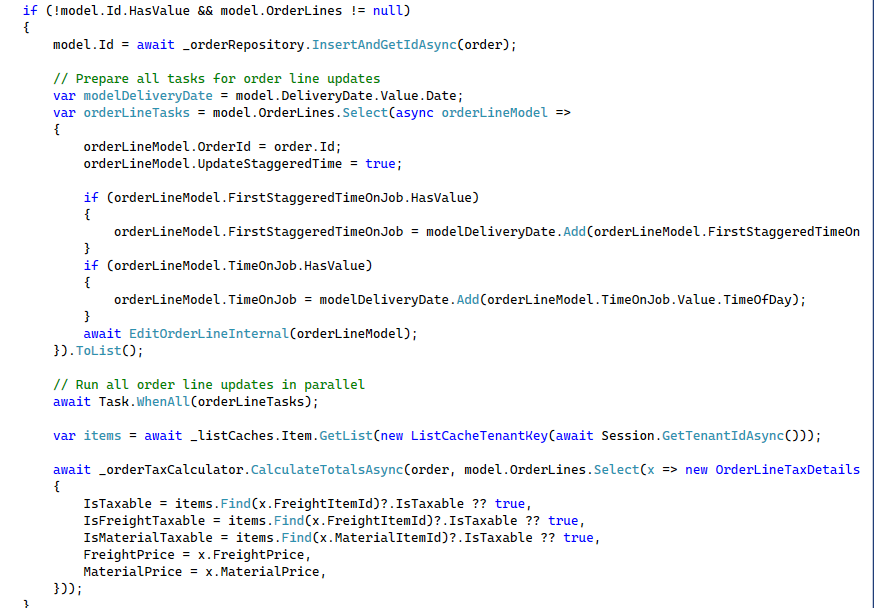
This method calls [EditOrder](#_EditOrder) which calls EditOrderInternal. Fixing EditOrderInternal may improve this. Note that there is some duplicated permission checking in these methods.

## EditOrder

This method calls EditOrderInternal which runs each orderline in a loop using await instead of running the updates in parallel.



vs

Are these updates batched when sending to SQL Server? Should AddRangeAsync be used to do a bulk operation?

## GetEditForOrderInternal

It is called from EditJob, GetOrderForEdit, and SyncLinkedOrders. Note in the image below that we are checking permissions multiple times. We also set up the permission check differently in each case. Should we do these permission checks once up front so we don’t duplicate them

A screenshot of a computer program

AI-generated content may be incorrect.

## GetOrders

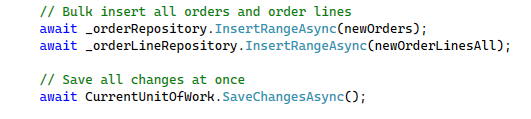
There aren’t any obvious issues here.

## GetQuotes

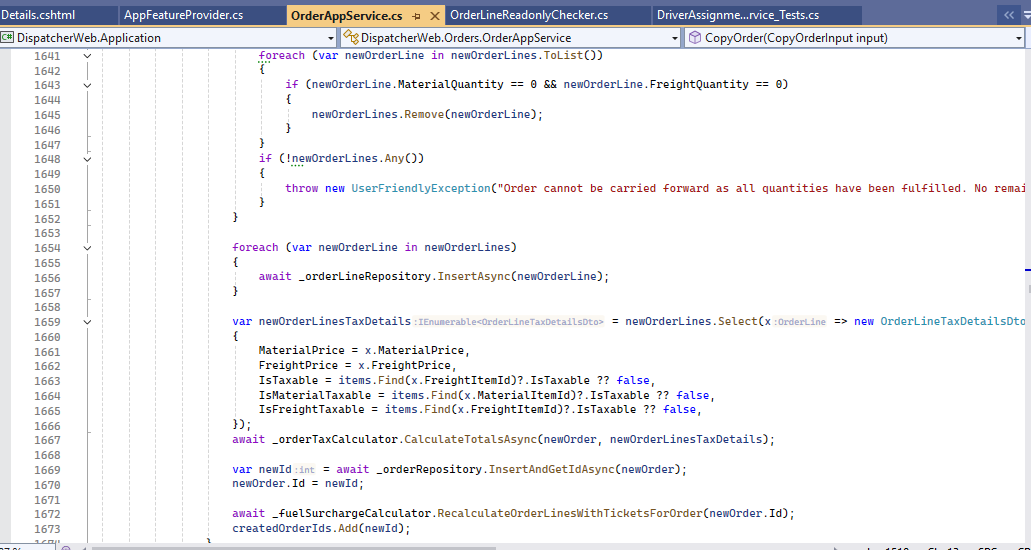
The recommendations are similar to GetQuotes.

1. Projecting to DTOs before paging and materialization: This avoids loading unnecessary navigation properties and reduces memory usage.
2. Applying all filters, sorting, and paging in the database: Prevents in-memory operations on large datasets.
3. Selecting only required fields: Reduces data transfer and query complexity.
4. Avoiding N+1 queries: By projecting related data in the initial query.

## CopyOrder

Insert all orderlines in as few calls to the database as possible and reduce the awaits inside a loop. Instead of the foreach at line 1654 as shown in the 3d image below, can you do something like this.

And then this



Don’t calculate the taxes and surcharges until after you have saved the other operations. This minimizes the time you hold the UOW open, potentially blocking other transactions.

## SetOrderLineIsComplete

To refactor SetOrderLineIsComplete in SchedulingAppService for better performance, focus on:

• Minimizing database round-trips: Batch queries and updates where possible.

• Avoiding per-entity awaits in loops: Update entities in memory, then save all changes at once.

• Bulk deleting and updating: Use batch deletes/updates for OrderLineTruck entities when possible.

• Projecting only needed fields:

## CancelDispatch

SendDispatchMessage

TicketListView

When querying related entities, select only the fields you need. CreateInvoicesForTickets

AddOrderLineTruck

GetInvoicePrintOut

## SetNotificationAsRead

Profile this method as it is involved in slow transactions. Are numerous updates being sent to SQL Server individually? If so, can we update these as a set (batch update) in EF? Or use a simple sproc without business logic where a list of Ids is passed in and they get updated as a set instead of looping and updating individual notifications.

A screenshot of a computer

AI-generated content may be incorrect.

## SetIsVerifiedForTickets

Profile this method as it is involved in slow transactions. Are numerous updates being sent to SQL Server individually? If so, can we update these as a set? This would be a simple sproc without business logic where a list of Ids is passed in and they get updated as a set instead of looping and updating individual tickets.

A computer screen shot of a computer code

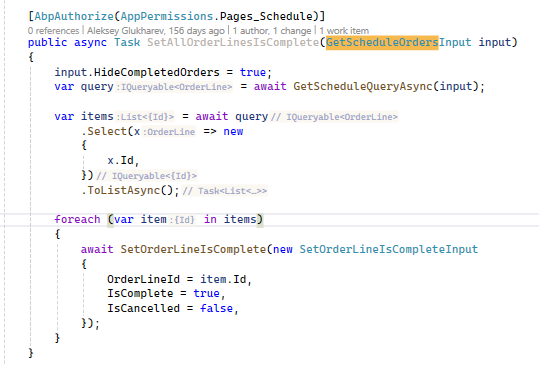
AI-generated content may be incorrect.

There are a lot of different data accesses being done in the CopyOrderTrucksInternal method as shown in the image below. I wonder if we should not worry about copying trucks any more and let them specify the trucks each day. Or do we need to optimize this significantly. It is called from CopyOrdersTrucks. Or should we at least simplify it by removing the code that checks utilization. Does setting the utilization to the same as the original gain anything for our customers. In other words, will it be the same from day to day. Or should this be a sproc where we don’t have multiple round trips between the app service and the database?

A screenshot of a computer program

AI-generated content may be incorrect.

SetAllOrderLinesIsComplete is frequently one of the methods with a long delay. Note how each line is set individually in a loop.



Inside the loop, the SetOrderLineIsComplete has multiple queries and complex logic. Could this be simplified or simplified enough to have a couple different paths depending on LeaseHauler vs dispatcher permissions and using a sproc?

Some tenants may have 100 orders being closed. Then multiply that by 5 queries in the SetOrderLineIsComplete which also could have multiple tickets being processed inside a loop. Meanwhile, even if there is no threadpool starvation yet, anything accessing any of these orders, orderlinetrucks, tickets, or dispatches may be getting locked at the database level until this method completes.

AddOrderLineTruck is another of those complex methods that is often involved in slow transactions. It runs >2000 times per day. A screenshot of a computer

AI-generated content may be incorrect.

It calls AddOrderLineTruckInternal that has most of the complex logic. Could we simplify this by removing the enforce validation setting and the code that returns the message. Also, remove the IsFreightPriceOverridden error. Aren’t we handling this short load problem differently now so that it will be OK to have 2 tickets? If not, what else could we change to not have this logic.

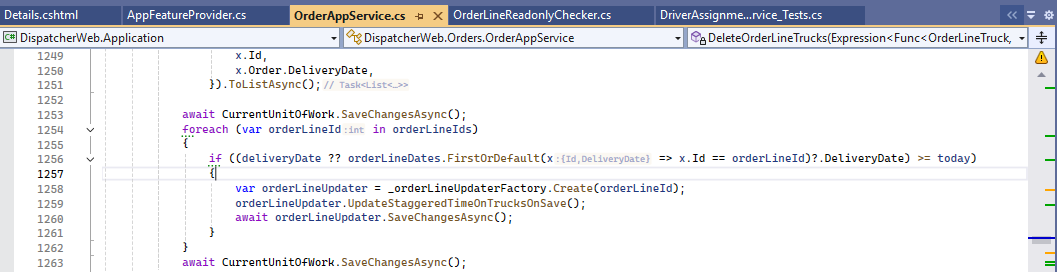
A computer screen shot of text

AI-generated content may be incorrect.

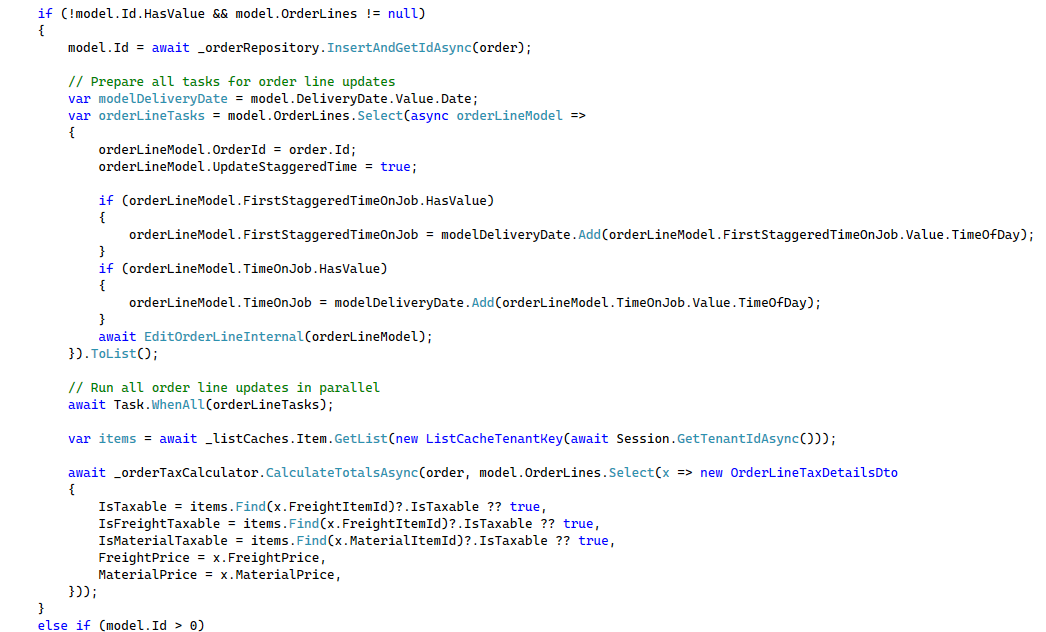
Should we create the driver assignment some way other than on assigning the truck, so that this additional insert/update doesn’t need to happen inside this logic.

## DeleteOrderLineTrucks

Instead of awaiting inside the loop, use parallelized await.



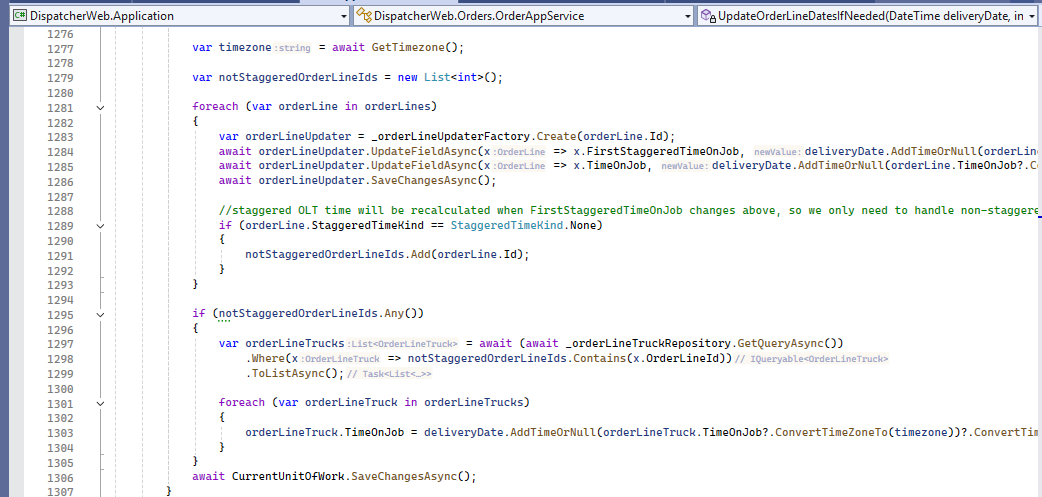
vs



Should AddRangeAsync be used to do as a bulk operation?

## UpdateOrderLineDates

Should we be awaiting these items in memory. They aren’t slow, so don’t need to be async , do they?

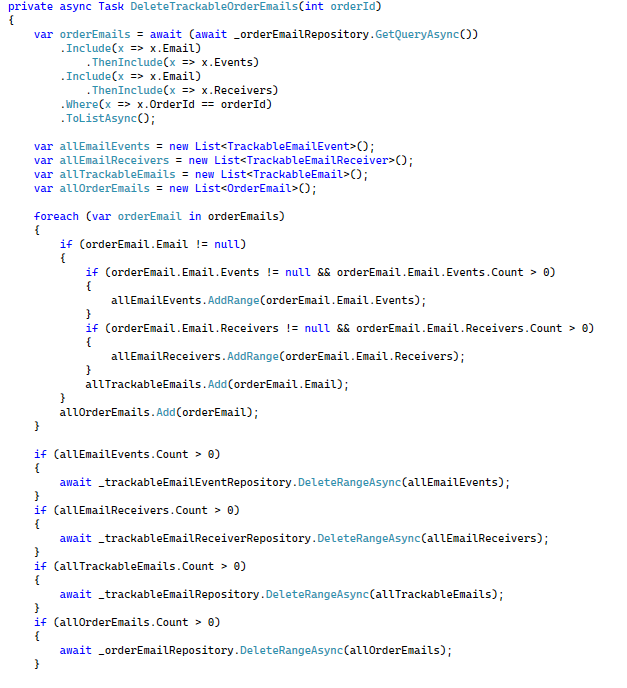


## DeleteTrackableOrderEmails

Avoiding await inside loops: Instead of deleting each event, receiver, and email one-by-one, collect all entities to delete and use batch deletes.

Minimizing database round-trips: Use DeleteRangeAsync for collections, and only call SaveChangesAsync once at the end.

Would something like this work better? Or if cascading deletes was set for referential integrity, this could be simplified further.



# Emails/TrackEvents

Looking in the TrackableEmails log, there are approximately 1100 of these per day. That works out to about 2 per minute across a normal 10 hour work day. This isn't a large amount. Maybe debouncing is all that is needed. This may prevent contention when a large number of events happen in a short period of time.

Note that the queries show up on the database list of long running queries.

A screenshot of a computer

AI-generated content may be incorrect.

# Reducing Database Transactions

Audit Logs.

Hangfire Logs. The sync requests result in a lot of insert transactions.

Further reduce RN App synch requests.

Reduce the PWA and RN Driver App payload by limiting the data queried based on the last synch time.

# Long Running Queries

A screenshot of a computer

AI-generated content may be incorrect.

**This is query 5668966 which is slow**

(@p0 datetime2(7),@p1 bigint,@p2 nvarchar(256),@p3 int,@p4 nvarchar(40),@p5 nvarchar(1000),@p6 nvarchar(100),@p7 bigint,@p8 uniqueidentifier,@p9 int)INSERT INTO [TrackableEmailEvents] ([CreationTime], [CreatorUserId], [Email], [EmailDeliveryStatus], [Event], [FailReason], [SendGridEventId], [SendGridEventTimestamp], [TrackableEmailId], [TrackableEmailReceiverId])

OUTPUT INSERTED.[Id]

VALUES (@p0, @p1, @p2, @p3, @p4, @p5, @p6, @p7, @p8, @p9)

**This is the query for 6850000 which is slow**

(@p0 datetime2(7),@p1 bigint,@p2 nvarchar(256),@p3 int,@p4 nvarchar(40),@p5 nvarchar(1000),@p6 nvarchar(100),@p7 bigint,@p8 uniqueidentifier,@p9 int,@p10 datetime2(7),@p11 bigint,@p12 nvarchar(256),@p13 int,@p14 nvarchar(40),@p15 nvarchar(1000),@p16 nvarchar(100),@p17 bigint,@p18 uniqueidentifier,@p19 int)MERGE [TrackableEmailEvents] USING (

VALUES (@p0, @p1, @p2, @p3, @p4, @p5, @p6, @p7, @p8, @p9, 0),

(@p10, @p11, @p12, @p13, @p14, @p15, @p16, @p17, @p18, @p19, 1)) AS i ([CreationTime], [CreatorUserId], [Email], [EmailDeliveryStatus], [Event], [FailReason], [SendGridEventId], [SendGridEventTimestamp], [TrackableEmailId], [TrackableEmailReceiverId], \_Position) ON 1=0

WHEN NOT MATCHED THEN

INSERT ([CreationTime], [CreatorUserId], [Email], [EmailDeliveryStatus], [Event], [FailReason], [SendGridEventId], [SendGridEventTimestamp], [TrackableEmailId], [TrackableEmailReceiverId])

VALUES (i.[CreationTime], i.[CreatorUserId], i.[Email], i.[EmailDeliveryStatus], i.[Event], i.[FailReason], i.[SendGridEventId], i.[SendGridEventTimestamp], i.[TrackableEmailId], i.[TrackableEmailReceiverId])

OUTPUT INSERTED.[Id], i.\_Position;

**Query 5661796** are the frequent Hangfire database entries. They aren’t slow but there are a lot of them.

# Miscellaneous Notes

Make sure the longer expiration time applies to the cached items that rarely change.



Fallback to updating caches in case redis fails.

Measure performance of Where vs WhereIf;

Measure performance of OrderBy(string);