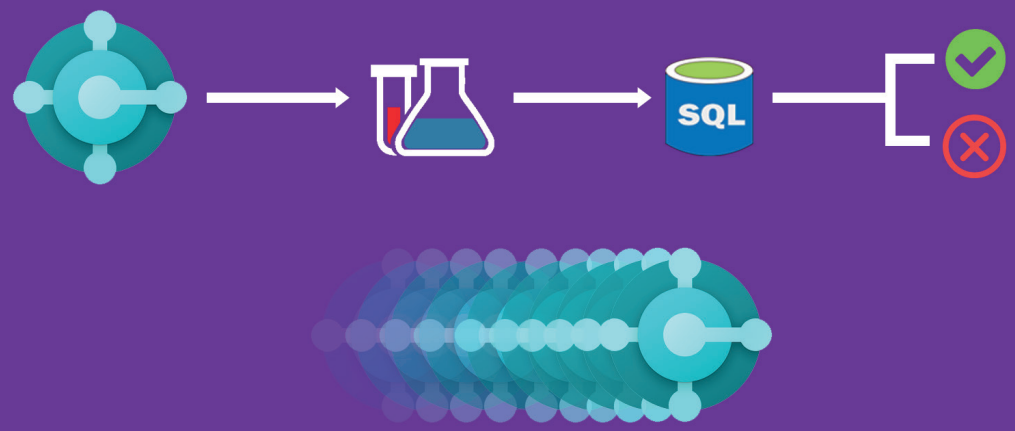
GUIDE: CODING FOR PERFORMANCE IN BUSINESS CENTRAL



Index

Introduction
One of the main problems with Business Central is performance, and while Microsoft is during all they can to try and optimize the application to be able to run faster. We, as developers, also have a responsibility to write our code correctly, and not just the way that we have always done. So in this short reference guide, I will try to cover some of the things that you should consider implementing when coding in AL.

Who is this guide for?
This guide is for anyone that is creating Extensions for Business Central. However, this guide is not a getting started with writing extensions guide, so before you start, you need to know the basics of writing AL code. I will also not cover how to setup Visual Studio Code or docker; this guide is 100% focused on how to improve the performance of your extensions.

Who am I?
My name is Dennis Fredborg, and I am at the time of writing this guide working as a senior developer at RelateIT. I have been writing blogs and creating videos about Microsoft NAV and Business Central, and all in between including Azure DevOps and docker, for a couple of years. I worked with Microsoft Dynamics since 2012, and before that, I worked with web and mobile development.

Increasing performance in general
There are some ways that you can increase your performance in a more general sense.

Best practice design patterns
One of the best ways to increase performance is to use best practice patterns. The reason why it is a good idea always to use best practice design patterns is well because they are the best practice. When using design patterns, you will make your code much easier to read by other developers, and on top of that, if you implement design patterns, you are sure that you structure your code in the best way possible.

Keep it simple
The title pretty much says it all; you should keep everything as simple as possible, this means stay away from adding unnecessary fact boxes, because while it might be nice to have all your information visible all the time, it will also result in a lot of background calculations. So, if there is some information that you want to display, consider offloading the calculations from the UI thread unto a **page background task** (we will get back to page background tasks later). In the same ballpark as fact boxes, you should also keep the number of cues on your role center to a minimum, and the number of flow fields on your pages should also be kept low. So, the whole idea is only to display, need to have, and not nice to have.

Create buffer tables

Instead of having expensive calculations running every time you open a page. Consider the fact if the data actually changes that much, or if you could set up a background task that could update the numbers perhaps once an hour, and then storing that number in a buffer table which you then can display on your Page.

Webservices

You should not expose standard pages as web services. There are a couple of reasons for this. The first is that if you expose a standard page, you do not have the same control over which fields are made available. The second is that many pages have fact boxes, which you should not display in web services because even though they will not be visible in the web service, Business Central will still execute them, which goes without saying it is just a waste of resources. The last reason is that many pages might have some code OnAfterGetCurrRecord, which might do a lot of UI changes but has no impact on what is exposed in the web service. So, like the fact boxes, it is a waste of resources. So, if you are using web service, you should always create a new page, so you have 100% control over your web services.

Built-in data types

To help increase performance, Microsoft has added some new data types that will not only make our life's easier but will also make our code run faster. The idea behind the new data types is that you can work with collections client-side and thereby decreasing the number of SQL transactions, which is one of the most significant performance issues in Business Central because the application is too tightly coupled with our database, that pretty much everything we do requires a call to the SQL server, while most other modern systems have a layer that will cache data and only writing data when needed. Microsoft knows that the tight coupling of the application and the SQL database is a performance problem, and they are working on moving more client-side; however, this is not an easy thing to do, but these new data types for sure is a step in the right direction.

TextBuilder

The TextBuilder data type is straightforward to use, and it was created to make working with strings easier. One of the places where you should use the TextBuilder is when you are concatenating strings, so instead of using += to add sting together, you should use the TextBuilder Append function. So, let us see how this works in practice:

```
var
    4 references
    myText: TextBuilder;

0 references
procedure AppendToText(myValue: Text)
begin
    myText.Append(myValue);
end;
```

As you can see it is pretty straight forward you define you TextBuilder and then use the Append function to add text to it, besides the Append function you also have the ability to add a new line using the AppendLine:

```
0 references
procedure AppendLineToText(myValue: Text)
begin
    myText.AppendLine(myValue);
end;
```

The difference between the two appends is that the append line will automatically add a line terminator to the end. You can also call the AppenLine without a value, which will result in it only adding the line terminator.
The TextBuilder also contains several ways to work with the text inside the TextBuilder, you, for example, have the Replace function:

```
0 references
procedure ReplaceText(oldVale: Text; newValue: Text)
begin
    myText.Replace(oldVale, newValue);
end;
```

This function will replace any value in the TextBuilder with a new value. You also have the Remove:

```
0 references
procedure Remove(startIndex: Integer; count: Integer): Boolean
begin
    myText.Remove(startIndex, count);
end;
```

Which will remove any substring from you TextBuilder, and then you have the ToText function:

```
0 references
procedure TotextString(): Text
begin
    Exit(myText.ToText());
end;
```

Which will convert whatever content you have in your TextBuilder to a text. Besides these, there are several other functions that you can use when working with the TextBuilder, but I will leave you to explorer them on your own.

Dictionary

The Dictionary data types are one of those datatypes that you have to master. The reason is that it is being used more and more across the Business Central Application. The reason being that it is an effortless way of storing structured data in memory. The way that the dictionary work is much in the same way that a real-world dictionary works, where you have a key, which would be your word, and a value which would be what your key means. As I wrote earlier, then dictionaries have become one of the preferred ways to pass data throughout Business Central, so let us look at how you can use the dictionary data type.
To add an item to your Dictionary, you must use the Add function:

```
0 references
procedure UsingDictionary()
var
    ValueDictionary: Dictionary of [Text, Text];
begin
    ValueDictionary.Add('Name','Bob');
    ValueDictionary.Add('Age','30');
end;
```

The signature of the Dictionary is Tkey, Tvalue, so you can create your Dictionary with any combinations of simple data types; in my example, I have created it with Text, Text. I then use the Add function to add new values to my Dictionary, and you can then later use the Get function to get the value of a given key by doing something like this:

```
0 references
procedure GetValueDictionary(ValueDictionary: Dictionary of [Text, Text])
var
    dictionaryValue: Text;
begin
    ValueDictionary.Get('Name',dictionaryValue);
end;
```

This function will copy the value of the associated key to your dictionaryValue variable; you also have the possibility to change the value by using the Set function.

```
0 references
procedure SetValueDictionary(ValueDictionary: Dictionary of [Text, Text])
begin
    ValueDictionary.Set('Name','Karl');
end;
```

If you wish to check if you Dictionary contains a given key, you can use the ContainsKey function:

```
0 references
procedure ContainsKeyValueDictionary(ValueDictionary: Dictionary of [Text, Text])
begin
    if ValueDictionary.ContainsKey('Name') then
        ValueDictionary.Set('Name','Karl');
end;
```

The ContainsKey will return a Boolean. You also have the ability to remove a key by using the Remove function:

```
0 references
procedure RemoveValueDictionary(ValueDictionary: Dictionary of [Text, Text])
begin
    ValueDictionary.Remove('Name');
end;
```

Should you choose to use an integer as your key you can create a for loop to loop through all your items:

```
0 references
procedure LoopDictionary(itemDictionary: Dictionary of [Integer, Code[20]])
var
    i: Integer;
    dictionaryValue: Code[20];
    Item: Record Item;
begin
    for i := 1 to itemDictionary.Count() do begin
        itemDictionary.Get(i, dictionaryValue);
        Item.Get(dictionaryValue);
    end;
end;
```

In this example, I have created a Dictionary where I used Integer as the key, which I incremented, and then I added the Item No. As the value, which I then can use to find my Items. Now, as I wrote earlier, then the Dictionary can only hold simple data types. So it can not contain records, however it can contain other Dictionaries which means you can indeed use the Dictionary as a replacement for your temporary tables you can see a simple way of doing this below:

```
You, a month ago | 1 author (You)
var
    1 reference
    itemDictionary: Dictionary of [Code[20], Dictionary of [Integer, Text]];

0 references
procedure CreateNestedDictionary(Item: Record Item)
var
    itemDic: Dictionary of [Integer, Text];
begin
    itemDic.Add(1, Item.Description);
    itemDic.Add(2, Item."Description 2");
    itemDic.Add(3, Item."Base Unit of Measure");
    itemDictionary.Add(Item."No.", itemDic);
end;
```

So, as you can see, then you have a lot of different possibilities with the Dictionary, which is why it is significant that Microsoft has given us this new data type to use.

List

The List is a straightforward data type, which you should use instead of Arrays and possibly also temporary tables. What makes the List data type better than the Array is that you do not have to define the size of your List as you do with your arrays. This means that the sky is the limit, besides that it works in much the same way that you would except a collection to work, you can define your List to be of any simple data type. You can use the Add function to add values to your collection:

```
var
    5 references
    itemList: List of [Text];
0 references
procedure CreateList(Item: Record Item)
begin
    itemList.Add(Item."No.");
end;
```

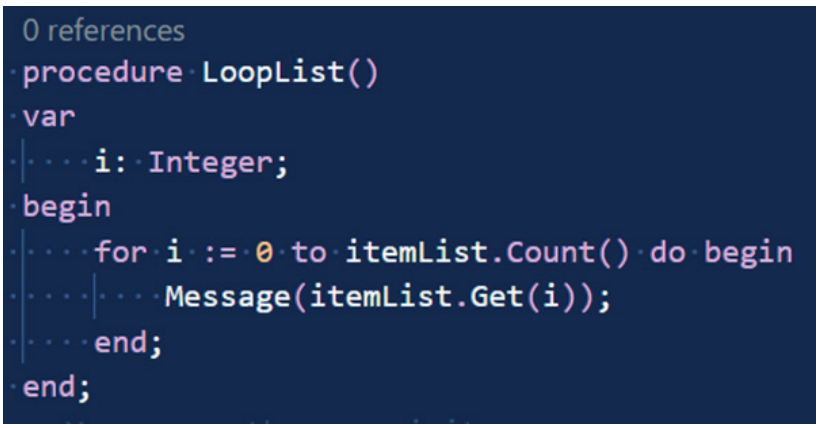
You can then use the Contains to check if a given value exists in your List:

```
0 references
procedure QueryList(itemNo: Code[20]): Boolean
begin
    if itemList.Contains(itemNo) then
        exit(true)
    else
        exit(false);
end;
```

You can use the Remove to delete items from your collection:

```
procedure DeleteItemFromList(ItemNo: Code[20])
var
begin
    itemList.Remove(ItemNo);
end;
```

And just like the Array, your collection has an index which means that you can use the for loop to loop through your collection and use the Get to retrieve the value:



So, as you can see, the List data type is a straightforward way to create a collection, with some great built-in functions to make it easier to work with your collection.

Conclusion

The conclusion of this chapter is, Microsoft has given us some new great built-in datatypes, which can help you to improve the performance of your extensions with little to no extra work. So next time you start writing your future extension, please keep these new data types in mind.

Multi-Threading

In Business Central, Microsoft has now given us the possibility to create a kind of multi-threading, which is fantastic; however, remember that with great power comes great responsibility. Multi-threading is one of those things that has always been missing in Microsoft Dynamics NAV, which was always disappointing, but do not get me wrong I understand why we haven't seen it before now. The reason is that once you make some multi-threaded, everything becomes much more complicated, which is also why we have only gotten a minimal type of multi-threading this time around, but maybe that will change in the future. So, why has Microsoft decided that now was the time to give us the ability to create multi-threading? The answer is quite simple, with all the complaints that Microsoft has gotten that the system runs slow when we navigate throughout the system. Microsoft had to find a way for us to offload our complicated and long-running tasks from our main UI thread, thereby not locking the user interface and forcing the end-user to wait until a given job has finished.

Page Background Tasks

Page background tasks are exactly what the name implies; they are a way for you to offload any reading task from your Page and move it to the background letting the user freely use the rest of the Page without having to wait. There is a couple of weaknesses when we are talking about page background tasks. The first is that you can only perform read operations in your background task, which means that you cannot write anything to the database. The second is that you can only have five background tasks running, should you start more than five they are queued, the third is that you must not change the record ID because this will cause the task to fail since it is bound to the record Id. The last is that the task only runs as long as the Page that started the task is open; once you close the Page, the task will be canceled. To get the most out of page background tasks, you should consider instead of creating flow fields, then you should create functions that can run in your background task, so the end-user does not need to wait for your flowfields to be calculated.
So let us see how you can create page background task, first, off you are going to need a codeunit. This codeunit must contain a variable of type Dictionary, which has to be of type Text, Text (I told you that Microsoft has started using Dictionary across the application). You can then execute whatever code you wish to run, and add the result to your Dictionary. The last thing you must do is call the Page.SetBackgroundTaskResult passing your Dictionary as a parameter:

```
codeunit 50100 MyPageBackgroundTask        You, 10 days
{
    trigger OnRun()
    var
        Result: Dictionary of [Text, Text];
        SalesLine: Record "Sales Line";
        Total: Decimal;
    begin

        // Run some code
        if SalesLine.FindSet() then
            repeat
                Total := Total + SalesLine.Amount;
            until SalesLine.Next() = 0;

        Result.Add('Total', format(Total));

        Page.SetBackgroundTaskResult(Result);
    end;

}
```

Next, you must add code to the Page that you wish to start the page background task. First off, you must create a global Integer that will hold the TaskID you do not need to set this ID as it will automatically be assigned when you start your background task on the OnAfterGetRecord trigger. You then execute the CurrPage.EnqueueBackgroundTask, where you pass the task ID, the codeunit that you wish to run, a Dictionary with wherever parameters that you might want to pass to the codeuint, a time out in milliseconds which defines for how long the task can run before it should fail. The last parameter is to determine at what level the task should fail. You also have to define a new trigger called OnPageBackgroundTaskCompleted which takes the TaskID and the Result Dictionary, in this function you can then act on the result of the background task, in my example, I am just updating a field on my Page:

```
0 references | Unsaved changes (cannot determine recent change or authors)
pageextension 50100 SalesOrderExt extends "Sales Order"
{
    Unsaved changes (cannot determine recent change or authors)
    layout
    {
        addlast(General)
        {
            field(Total; Total)
            {
                Editable = false;
                ApplicationArea = all;
            }
        }
    }

    Unsaved changes (cannot determine recent change or authors)
    var
        2 references
        Total: Text;
        2 references
        WaitTaskId: Integer;

    trigger OnAfterGetRecord()
    var
        //Defines a variable for passing parameters to the background task
        TaskParameters: Dictionary of [Text, Text];
    begin
        //TaskID, CodeunitId, Parameters, Timeout, ErrorLevel
        CurrPage.EnqueueBackgroundTask(WaitTaskId, 50100, TaskParameters, 1000, PageBackgroundTaskErrorLevel::Error);
    end;

    trigger OnPageBackgroundTaskCompleted(TaskId: Integer; Results: Dictionary of [Text, Text])
    begin
        if (TaskId = WaitTaskId) then begin
            Total := Results.Get('Total');
        end;
    end;
}
```

And that is pretty much all there is to page background tasks, so as you can see, they are effortless to use. If you use them correctly, they can help you to give the end-user a better experience, and they are quite harmless since there is no cleanup. Since you cannot perform write operations, you are not going to end up creating deadlocks, which is one of the many problems with multi-threading.

Background Sessions

The background session is more potent than the page background task because it comes with very few limits; however, as the name suggests, it will indeed create a new session, which is very costly. So, you should only use Background Sessions when the code that you wish to execute is costly. The way that Background sessions work is much the same way that you could do in NAV, where you let your code call a self-published web service. It has the same limits that web services do, which is that it cannot handle UI's, and a crucial thing when working with Background Sessions is to remember to stop them once you are done; otherwise, they will not die. The code to start a Background Session is very simple. You create a variable of type Integer that will hold your session ID. You then call the STARTSESSION with the session ID and the codeunit that you wish to execute. You can also pass a CompanyName and a record to the STARTSESSION, the STARTSESSION will return a Boolean, which you can use to determine rather or not you need to close the session again:

```
var
    SessionID: Integer;
    OK: Boolean;
begin
    OK := STARTSESSION(SessionId, CODEUNIT::RunBackgroundSession);
    if OK then
        STOPSESSION(SessionId, 'Logoff Item Description session')
    else
        ERROR('The session was not started successfully.');

end;
```

So the Background Sessions are a compelling way of creating a new thread because you can pretty much do anything in a Background Session. However, this also comes with a price, the main one being that you will create a whole new session on your tenant, and it is hard to control the Background Session once it has started, and because you can do write transactions in the Background Session, you can cause deadlocks.

Task Scheduler

The Task Scheduler is by no means anything new in NAV / BC, but it is still important to note that this is also an option if you wish to create a background thread, but there is nothing new here it works as it has always done:

```
trigger OnAction()
var
    JobQueueEntry: Record "Job Queue Entry";
    OK: Boolean;
begin
    Ok := TaskScheduler.CanCreateTask();

    if OK then begin
        TASKSCHEDULER.CREATETASK(Codeunit::TaskScheduler, CODEUNIT::"Job Queue Error Handler", TRUE, COMPANYNAME);
    end;

end;
```

You should consider using this option instead of the Background Session because you have more control when it comes to the Task Scheduler, while, of course, the big downside is that it does rely on the job queue. Unlike both the Page Background Task and the Background Sessions, the Task Scheduler survives server restarts.

Conclusion

Microsoft has now given us some tools to help improve performance by offloading tasks to other threads, and I would highly recommend that you investigate implementing the Page Background Task wherever it makes sense. In contrast, the other two types of tasks you should use with caution because you might end up doing more harm than good.

Unit Testing Performance

One of the best weapons against slow performance is testing and monitoring. The primary way of testing is doing manual testing, where you would go through the application and get a real-world view of how everything runs; however, this can be a very daunting task. The problem is that something can run fast one day, and then a change can be made to the system that makes it run slow the next, and also manual testing for performance can quickly become subjective because something someone finds slow someone else might not find slow. So, while you should always perform manual testing, it cannot stand alone. The next thing you can do is set up Application Insight. Which can be used to monitor Business Central as a whole, and this is very easy to set up, all you need to do is create an Application Insights on your Azure subscription, this will give you a key that you can add to your tenants in your Partner Center. The service will then start feeding your Applications Insight with all the data that you would find in the only days in your SQL profiler. I would suggest that you set this up on the customers that you wish to monitor. However, this is not a way to catch performance issues upfront, but more a way to monitor when things are already running slow in production. The last way is to create performance unit tests that, just like any other unit tests, are used to be able to catch any errors and performance issues before they make it to production. So, let us see how you can get started with writing performance unit tests. Microsoft has given us the ability to call the SessionInformation, which has two methods. SqlRowsRead, which will return the number of rows read throughout the whole session. SqlStatementsExecuted() will return the number of statements executed throughout your session; both of these numbers can also be found in your debugger. However, being able to get this information through code gives us the ability to run unit tests on the results.

SqlRowsRead

To run the unit test on your SqlRowsRead, you must first create a codeunit with subtype Test, since the SqlRowRead will return all the rows read throughout your entire session. We are only interested in the rows read doing or test. So, we need to run the SqlRowsRead before and after our code and get the difference, next you need to figure out how many reads are too many for your code and test rather it stays within the limit, an example could be as follows:

```
codeunit 50100 "TestingPerformanceCodeunit"
{
    Subtype = Test;

    [Test]
    0 references
    procedure SQlReadTestReadLimitGet()
    var
        Customer: Record Customer;
        SqlRowsReadBefore: BigInteger;
        SqlRowsReadAfter: BigInteger;
    begin
        SqlRowsReadBefore := SessionInformation.SqlRowsRead();
        Customer.Get('10000');
        SqlRowsReadAfter := SessionInformation.SqlRowsRead();
        if SqlRowsReadAfter - SqlRowsReadBefore > 1 then
            Error('Too many reads: ' + format(SqlRowsReadAfter - SqlRowsReadBefore));
    end;
```

This code check that when I use a Get on my customer, that I only make one read operation, I know this is a silly example, but the reason why I use it is to keep it as simple as possible.

SqlStatementsExecuted

The SqlStatementsExecuted works the same way as the SqlRowRead, with the only difference that it gives you the number of SQL stamens that were executed, again here is a simple example.

```
[Test]
0 references
procedure SqlExcecuteValidate()
var
    Customer: Record Customer;
    SqlExecutedBefore: BigInteger;
    SqlExecutedAfter: BigInteger;
begin
    SqlExecutedBefore := SessionInformation.SqlStatementsExecuted();
    Customer.Get('10000');
    Customer.Validate(Name, 'NewNameValidate');
    Customer.modify(true);
    SqlExecutedAfter := SessionInformation.SqlStatementsExecuted();
    Error(format(SqlExecutedAfter - SqlExecutedBefore));
end;
```

I will not go into more detail since I feel that testing is too broad a topic for this short guide, but I just wanted to let you know that you can indeed test the performance of your code.

Conclusion

Microsoft has given us some strong tools to help us verify that our code runs as optimal as possible. Still, it is essential to remember that there is no silver bullet, just because you start adding performance unit tests. Your code will not magically be 100% optimized, but it is a great tool to add to your arsenal, to iron out the most significant performance problems. While the Application Insight can help you identify bottlenecks, sometimes those bottlenecks cannot be fixed, either because it lies in the base app or in a third-party extension, where you have no power to change anything. So, the best you can do is make sure that your code is as optimized as it can be.

Finale notes

As you can see then, Microsoft is trying to give us some tools to help us create more performance-friendly extensions, which is a step in the right direction, of course, there are still many performance problems that are out of our hands, and we can only hope that things get better very soon. One of the main things that I would like to see Microsoft fix is the table extensions issue. For those of you that do not know about this issue. The problem is that every time you create a table extension, then it will automatically create a new table; which is linked to the main table; which means that when you, for example, want to read the Item table it will have to create a SQL join behind the scenes on all the tables of the table extensions. And while Microsoft's answer to this at the moment is that we should limit our usage of table extensions and instead create supporting tables, this is not a valid answer, because this will over-complicate our code. The sad thing is that specifically, table extensions was one of the coolest features of AL, so I am keeping my fingers crossed that Microsoft will come up with a better solution.

There are a lot of performance topics that I did not cover in this guide, but I will likely get to those topics at another time, but that is the issue with creating blogs, videos or guides/books at the moment, our world is changing so fast at the moment that you cannot include everything, because then you will never be done. Still, do not get me wrong I find that excellent because in that way I am always kept on my toes. Well, that is it for now. I hope you found some useful information in these pages, and until next stay safe.