Microsoft Dynamics® AX 2012

Best practices for developing customizations

White Paper

This document describes best practices and recommended patterns and practices for developing customizations for Microsoft Dynamics AX 2012.

October 2012

microsoft.com/dynamics/ax

Mudit Mittal and Arijit Basu, Senior Solutions Architects

Table of Contents

Introduction

This document describes best practices and recommended patterns and practices for developing customizations for Microsoft Dynamics AX 2012. The document describes common coding scenarios and best practices, and provides examples of common errors and how to avoid them.

Overview

When you design and develop applications in Microsoft Dynamics AX 2012, there are key patterns that you must consider to ensure scalable, maintainable, and high-performing applications. The following illustration shows the critical steps that these patterns are within.

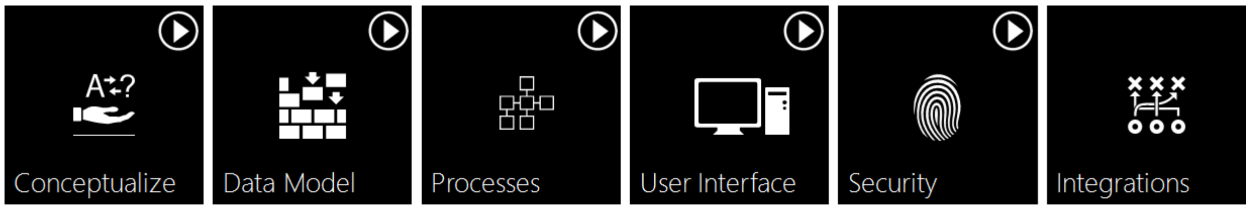


Figure 1 Steps in developing an application

Within each critical step, you will perform tasks illustrated in the following diagram. Hold down the Ctrl key and click each section to view details in that section.

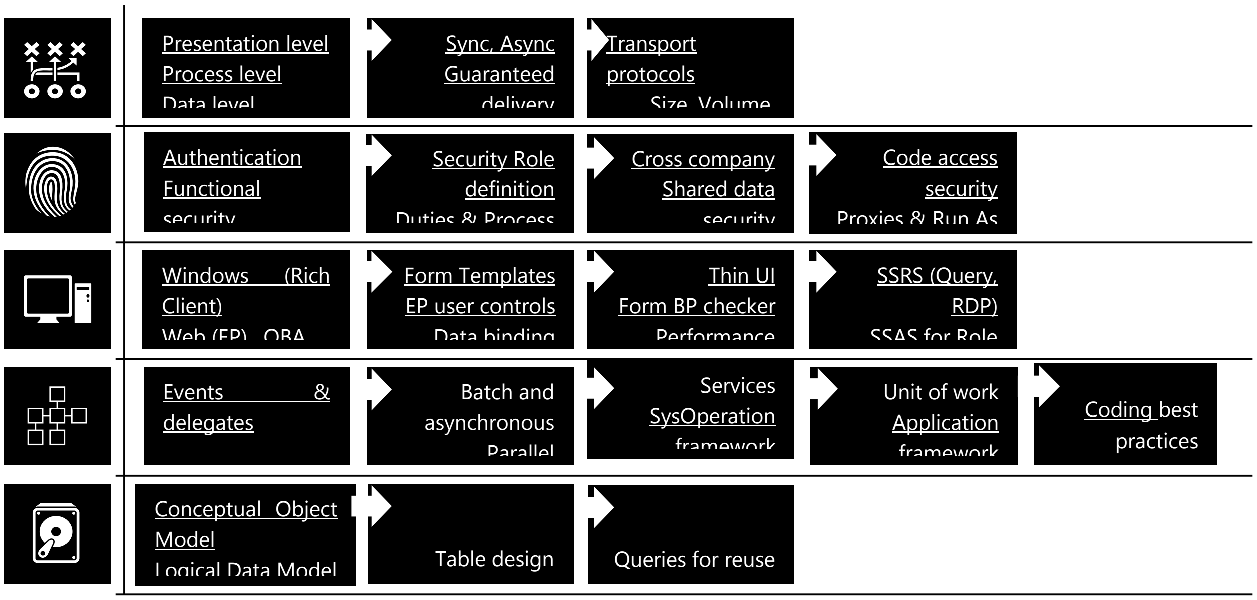


Figure 2 Tasks associated with each step

This white paper describes the patterns when you store, process, and visualize the data.

- **Store the data** – You can design and write code to store business data. You store and retrieve data by developing and modifying tables, data types, and related objects. The data model is important to the application. An efficient data model will improve performance, maintenance, and the business value proposition.
- **Process the data** – You can process and manipulate the creation and extraction of business data. You will convert data into information by running it through business logic and business parameters. Processing data is usually written in classes. When you write effective business logic, your application will be reusable, scalable, and high-performing.
- **Visualize the data** – You can capture and present data on different UI elements – for example, Microsoft Windows, the web, and devices. When you follow the UX guidelines, you increase the usability of your application.

Conceptualize

The first step is to conceptualize the application. You must understand the problem you want to solve.

One of the basic concepts when developing applications in Microsoft Dynamics AX is reuse. You may reuse functionality when you make a new application, or when you modify or extend an existing application. If you will add new functionality to an existing application, know the individual parts of the existing application. If you do not understand the existing application pattern, you may create functionality that was not necessary, or that already exists in the application.

Data model

Data modeling is a process used to analyze and define data requirements for the following purposes:

- Designing databases that support business requirements
- Integrating information systems

The following sections describe the three types of data models.

Conceptual object model

A conceptual object model (COM) is a technology-agnostic definition of the data and relationships.

Logical data model

A logical data model (LDM) is a representation of the business data. It is organized in terms of entities and relationships. The LDM is generally created during the detailed specification phase. The purpose of the LDM is to represent the relevant business requirements relationally in BCNF (Boyce-Codd normal form). The LDM elaborates the COM structure to a level of detail where all required relation types (entity types and relationship types), relationships, and attribute types are represented.

Physical data model

A physical data model (PDM) defines the details of the data design. It includes elements such as tables and indexes. The PDM considers a specific database management system (DBMS) and is generally created in conjunction with the Development Design document. The PDM is derived from the LDM. You can denormalize the model to improve performance. The denormalized PDM must implement the business requirements represented in the LDM.

Table design

The following sections describe best practices that apply to data types, table designs, and key and index definitions.

Temporary tables – InMemory vs. TempDb

The choice of whether to use a TempDB or InMemory temporary table depends on the scenario where it is used.

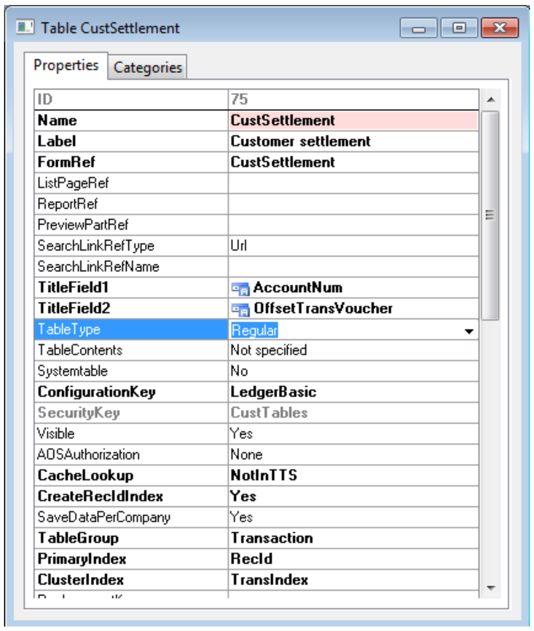
Usually, TempDB temporary tables are faster than in InMemory ones. They are maintained in the database and support joins with regular tables.

InMemory temporary tables are instantiated in the active memory of the tier the process is running on. The process can run on the client tier or the server tier. The objects are held in memory until the size reaches 128 KB. The dataset is then written to a disk file on the server tier. You can use InMemory temporary tables when the amount of data is small and Microsoft SQL Server round trips should be avoided.

Both InMemory and TempDb tables can be used as data sources on forms.

Note: An InMemory table cannot be joined in an X++ SQL statement with Regular/TempDB tables. It must be the outer table in a join. It is better to use TempDB temporary tables, because SQL operations are faster due to being maintained in the database.

In Figure 3, you can see the **TableType** property for a table, where you can set the type as **Regular**, **InMemory**, or **TempDB**.



ID	75
Name	**CustSettlement**
Label	**Customer settlement**
FormRef	**CustSettlement**
ListPageRef	
ReportRef	
PreviewPartRef	
SearchLinkRefType	Url
SearchLinkRefName	
TitleField1	**AccountNum**
TitleField2	**OffsetTransVoucher**
TableType	Regular
TableContents	Not specified
Systemtable	No
ConfigurationKey	**LedgerBasic**
SecurityKey	**CustTables**
Visible	Yes
AOSAuthorization	None
CacheLookup	**NotInTTS**
CreateRecIdIndex	**Yes**
SaveDataPerCompany	Yes
TableGroup	**Transaction**
PrimaryIndex	**RecId**
ClusterIndex	**TransIndex**

Figure 3 TableType property

Example scenario

In this example, data needs to be inserted into a temporary table that is based on a regular table. There are a large number of records in the regular table.

Bad solution: Regular table is mixed with an InMemory table, using row-wise operations

Using an InMemory table to solve this problem is not a good idea, because records are inserted into InMemory temp tables row by row. Using row-wise operations to insert data into the temporary table would lead to many iterations of the loop.

Also, the second **while** loop would not work if the temporary table type is **InMemory**, because it would involve joining a temporary table with regular tables.

```
static void tempTableMixedWithRegularTable(Args _args)
{
    TmpSalesItemReq tmpSalesItemReq;
    InventTable     inventTable;
    PriceDiscTable  priceDiscTable;
    SalesTable      salesTable;
    SalesLine       salesLine;

    while select ItemId, ProjCategoryId from inventTable
    {
        /* temporary table initialized from regular table*/
        tmpSalesItemReq.Itemid = inventTable.Itemid;
        tmpSalesItemReq.ProjCategoryId = inventTable.ProjCategoryId;
        /* Initialize other fields */
        tmpSalesItemReq.doInsert();

    }

    while select SalesId, DocumentStatus from salesTable
            where salesTable.DocumentStatus == DocumentStatus::Confirmation
        join SalesId, ItemId from salesLine
            where salesLine.SalesId == salesTable.SalesId
        /* temporary table joined with regular table*/
        join ItemId, SalesPrice, SalesQty, SalesUnit from  tmpSalesItemReq
            where tmpSalesItemReq.ItemId == salesLine.ItemId
        join priceDiscTable
            where priceDiscTable.ItemCode == TableGroupAll::Table
                  && priceDiscTable.ItemRelation == tmpSalesItemReq.itemId
    {
        /* Process */
    }
}
```

Good solution: Regular table is mixed with a TempDB table

The following example demonstrates how to solve the problem by changing the temporary table type to **TempDb** (instead of **InMemory**). Records are inserted into a TempDB temp table by using a set-based API (**insert_recordset**). Using **insert_recordset** inserts multiple records in a single set-based operation.

```
static void tempdbTableMixedWithRegularTable(Args _args)
{
    TmpSalesItemReq tmpSalesItemReq;
    InventTable     inventTable;
    PriceDiscTable  priceDiscTable;
    SalesTable      salesTable;
```

```
        SalesLine       salesLine;

        /* temporary table initialized from regular table*/
        insert_recordset tmpSalesItemReq(ItemId, ProjCategoryId)
            select ItemId, ProjCategoryId from inventTable;

        while select SalesId, DocumentStatus from salesTable
                where salesTable.DocumentStatus == DocumentStatus::Confirmation
            join SalesId, ItemId from salesLine
                where salesLine.SalesId == salesTable.SalesId
            /* temporary table joined with regular table*/
            join ItemId, SalesPrice, SalesQty, SalesUnit from  tmpSalesItemReq
                where tmpSalesItemReq.ItemId == salesLine.ItemId
            join priceDiscTable
                where priceDiscTable.ItemCode == TableGroupAll::Table
                      && priceDiscTable.ItemRelation == tmpSalesItemReq.itemId
        {
            /* Process */
        }
    }
```

Time zones and date effectivity

You can use the **UtcDateTime** field to store date and time values. It has support for different time zones. Date fields should not be used unless they represent a value that is independent of the time zone. For example, use a date field for a birth date or an anniversary.

If the table has **validFrom** and **validTo** attributes, the table should be marked as a valid time state table. To mark the table as a valid time state table, set the **ValidTimeStateFieldType** property to **UTCDateTime** or **Date**. The **Date** field and **utcDateTime** field then implement date effectivity correctly.

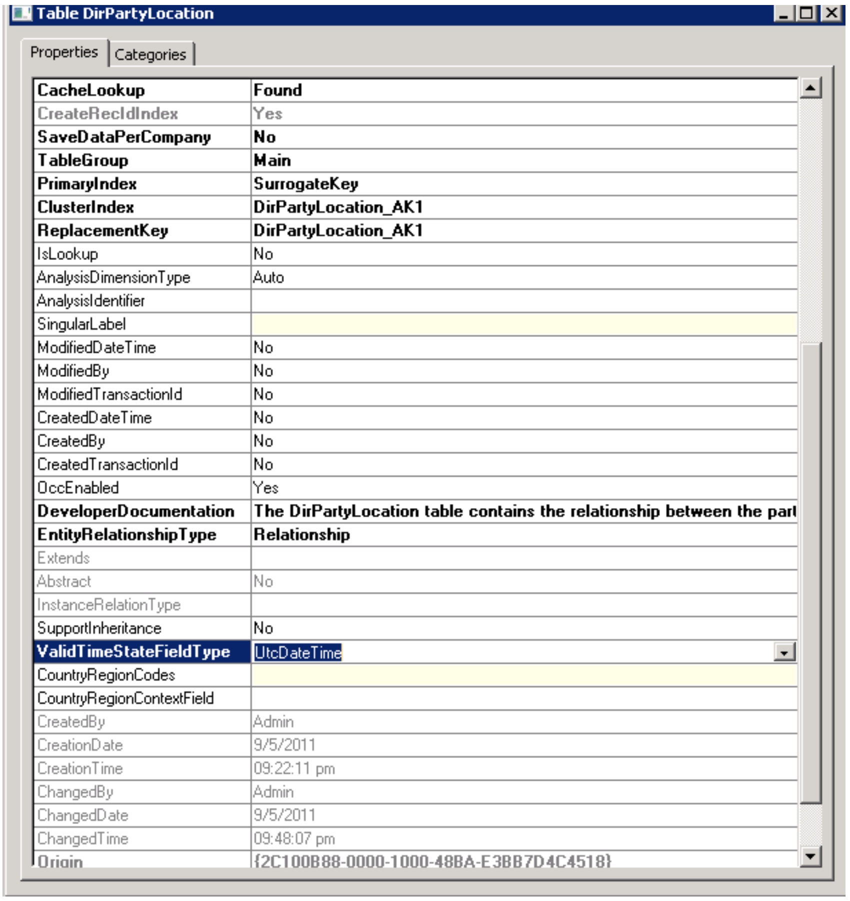


Table DirPartyLocation

Properties | Categories

CacheLookup	**Found**
CreateRecIdIndex	Yes
SaveDataPerCompany	**No**
TableGroup	**Main**
PrimaryIndex	**SurrogateKey**
ClusterIndex	**DirPartyLocation_AK1**
ReplacementKey	**DirPartyLocation_AK1**
IsLookup	No
AnalysisDimensionType	Auto
AnalysisIdentifier	
SingularLabel	
ModifiedDateTime	No
ModifiedBy	No
ModifiedTransactionId	No
CreatedDateTime	No
CreatedBy	No
CreatedTransactionId	No
OccEnabled	Yes
DeveloperDocumentation	**The DirPartyLocation table contains the relationship between the part**
EntityRelationshipType	**Relationship**
Extends	
Abstract	No
InstanceRelationType	
SupportInheritance	No
ValidTimeStateFieldType	UtcDateTime
CountryRegionCodes	
CountryRegionContextField	
CreatedBy	Admin
CreationDate	9/5/2011
CreationTime	09:22:11 pm
ChangedBy	Admin
ChangedDate	9/5/2011
ChangedTime	09:48:07 pm
Origin	{2C100B88-0000-1000-48BA-E3BB7D4C4518}

Figure 4 ValidTimeStateFieldType property

Base layer

This section describes best practices related to the base layer tables. You may want to store additional information in existing entities that are modeled in the base layer tables.

Do not add a large number of fields to the base tables.

Normalize any new schema that you add.

Consider adding required fields to a new table that has a foreign key (FK) relationship to the base table. Putting fields directly in a base table has many disadvantages, including the following:

- Query and maintenance costs increase, because the new fields are not required for all scenarios.
- For future versions, if the base layer table is modified or removed, the upgrade cost to move those fields increases. If new fields are stored in a different table, upgrade can be handled easily by changing the FK in the customized table.

Surrogate key

Use a surrogate key as the primary key and the foreign key.

Avoid using a natural key as the foreign key, because the natural key is attached to the business, and the value for it may change over time as required by the needs of the business. If a natural key is referenced, a change in its value would require updating all references. Surrogate keys are not attached to the business, and even if the natural key changes, the references to the surrogate key do not need to be updated.

Data type

Use extended data types (EDTs) to create new fields, or to define parameters or variables in code.

Avoid using base data types such as **int**, **str**, and **str30**, which have the following disadvantages:

- If you reference the data type in code in multiple places, the cost of updating code is high.
- It is easy to introduce inconsistency in usage. For example, you could use **str** instead of **str30**.

In case of country-specific or region-specific functionality, verify that the country/region context is set correctly in all places, such as fields and EDTs.

Enforce constraints in the database schema. When constraints are defined in code, it is easier to miss enforcing a constraint and therefore end up with inconsistent data. Design the schema without influence from the UI.

In the case of a complex or fully normalized schema, we recommend that you create views for use in the UI and reporting. For example, by default, Microsoft Dynamics AX ships with views for dimensions and addresses.

In some scenarios denormalization might also be needed. If denormalization is required, keep the following guidelines in mind:

- Factor infrastructure and performance limitations into your design.
- If denormalization is required, your design will include redundant data. Ensure that all redundant data is kept in sync.

Base enums

Do not change existing enum values. Modifying base enum values will have a significant upgrade impact. If you have modified base enum values, you must write and test the required upgrade scripts.

If you plan to create new enum values in a standard enum, leave a gap between existing and new values. The gap should be left so that your custom values do not conflict with new enum values that are added by add-ons or version upgrades. If there is a conflict, you must create upgrade scripts.

Table properties

When you create a new table, set the relevant table properties in the property sheet. Incorrect table property settings can negatively affect performance and manageability. The following screenshot illustrates the properties to set for a table.

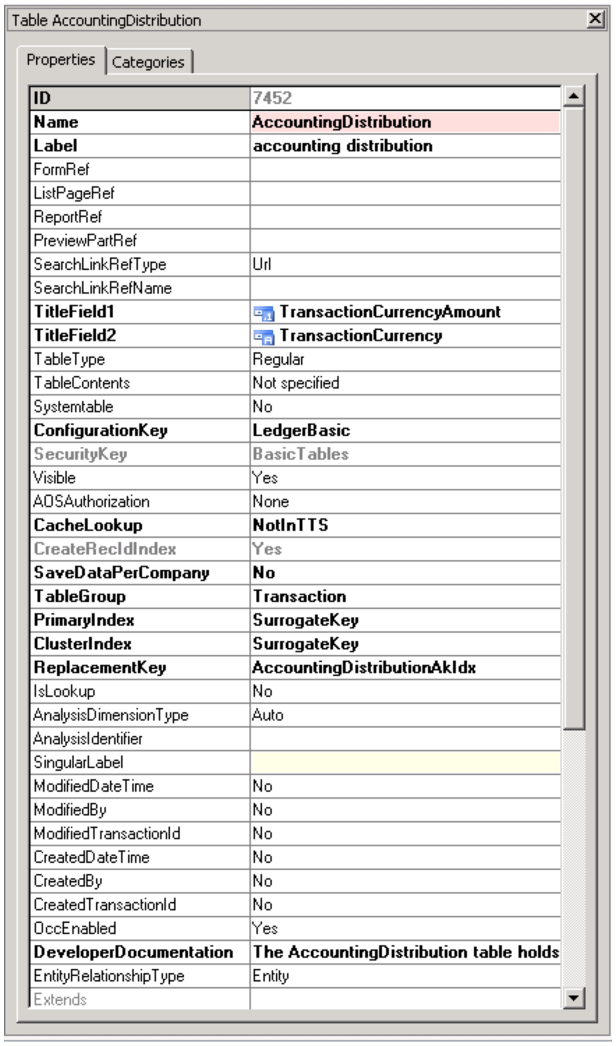


Table AccountingDistribution

Properties | Categories

ID	7452
Name	AccountingDistribution
Label	accounting distribution
FormRef	
ListPageRef	
ReportRef	
PreviewPartRef	
SearchLinkRefType	Url
SearchLinkRefName	
TitleField1	TransactionCurrencyAmount
TitleField2	TransactionCurrency
TableType	Regular
TableContents	Not specified
Systemtable	No
ConfigurationKey	LedgerBasic
SecurityKey	BasicTables
Visible	Yes
AOSAuthorization	None
CacheLookup	NotInTTS
CreateRecIdIndex	Yes
SaveDataPerCompany	No
TableGroup	Transaction
PrimaryIndex	SurrogateKey
ClusterIndex	SurrogateKey
ReplacementKey	AccountingDistributionAkIdx
IsLookup	No
AnalysisDimensionType	Auto
AnalysisIdentifier	
SingularLabel	
ModifiedDateTime	No
ModifiedBy	No
ModifiedTransactionId	No
CreatedDateTime	No
CreatedBy	No
CreatedTransactionId	No
OccEnabled	Yes
DeveloperDocumentation	The AccountingDistribution table holds
EntityRelationshipType	Entity
Extends	

Figure 5 Table properties

Virtual company

Avoid using custom table collections. If custom table collections are necessary, run best practices on the environment after you update cross-references. To implement virtual companies correctly, there should be no best practice checks on the table collection, and all FK references must be resolved in the same table collection.

Table inheritance

Table inheritance should be used mainly for entities and should not be used for transaction tables.

For example, for entities, such as Party, table inheritance is useful. However, a transaction for a person or organization is not a good match for table inheritance.

Table metadata

To ensure best performance, it is important to set table properties appropriately.

Table group

Set the **TableGroup** property on tables correctly. Use the following guidelines to decide the correct table group property.

The following table describes the table group definitions and provides examples.

Table group	Table has these characteristics	Examples
Parameter	The table contains data primarily used as parameters or setup information for one of the main tables (a table that has a **TableGroup** property of **Main**). The table typically contains only one record per company.	CustParameters, VendParameters
Group	The table contains data primarily used to categorize one of the main tables (a table that has a **TableGroup** property of **Main**). There is a one-to-many relationship between a **Group** table and a **Main** table.	CustGroup, VendGroup
Main	The table is one of the principal tables in the application and contains data for a central business object. The table typically holds static, base information. There is a one-to-many relationship between a **Main** table and a **Transaction** table.	CustTable, VendTable
Transaction	The table contains transaction data. The table is typically not used directly for data entry. It is imperative that you not delete data in these tables without applying appropriate archiving rules, because deleting from them might affect the integrity of the data.	CustTrans, VendTrans
Worksheet Header	The table typically categorizes information in the **WorkSheetLine** tables. There is a one-to-many relationship between a **WorkSheetHeader** table and a **WorkSheetLine** table.	SalesTable
Worksheet Line	The table contains information to be validated and made into transactions. In comparison to the data contained in a **Transaction** table, the data in **WorkSheetLine** tables is temporary and may be deleted without affecting system stability.	SalesLine
Miscellaneous	The table does not fit into any of the other categories. This is the default value for a new table.	TableExpImpDef

Optimistic concurrency control

Use optimistic concurrency control (OCC) unless you have a business reason for using pessimistic control.

OCC helps increase database performance. Pessimistic concurrency control locks records as soon as they are fetched from the database for an update. However, optimistic concurrency control only locks records from the time when the actual update is performed.

Pessimistic concurrency control was the only option available in Microsoft Axapta 3.0 (now a part of Microsoft Dynamics). You can now choose which concurrency model to use – optimistic or pessimistic.

The following are the advantages of using OCC:

- Fewer resources are used to hold the locks during the update process.
- Records are locked for a shorter length of time.
- Records remain available for other processes to update if they have been selected from the database but haven't yet been updated.

The disadvantage of using OCC is that the update can fail if another process updates the same record. If the update fails, it must be retried. This can lead to a reduction in database performance.

Cache lookup

Be sure to set the **CacheLookup** property on tables correctly. If the property is set to **None**, the system cannot take advantage of caching for that table. Choose the property value based on the behavior and use of the table. Use the following guidance to determine the appropriate settings for a table:

1. Determine the table group of the table.
2. Use one of the appropriate cache lookups (see the following table).
3. Make sure that the table has a relevant primary key.
4. Make sure that a unique key is set for tables, where one exists, to leverage unique index caching.

The following table describes the appropriate cache lookups for table groups.

Table group	Appropriate cache lookups for this table group	Inappropriate cache lookups for this table group	Reason for recommendations
MAIN	FOUND, FOUND&EMPTY	NOTINTTS, ENTIRETABLE	Mainly for master data. NotinTTS does not apply unless data needs to be reread within a transaction block.
GROUP	FOUND, FOUND&EMPTY, EntireTable	NOTINTTS	Avoids an unnecessary read within the transaction block.
PARAMETER	FOUND, FOUND&EMPTY, EntireTable	NOTINTTS	This data should not change often.
WORKSHEETHEADER	NotInTTS, NONE	FOUND, FOUND&Empty, ENTIRETABLE	Data needs to be reread within the transaction block.
WORKSHEETLINE	NotInTTS, NONE	FOUND, FOUND&Empty, ENTIRETABLE	Data needs to be reread within the transaction block.
TRANSACTION	NotInTTS, NONE	FOUND, FOUND&Empty, ENTIRETABLE	Data needs to be reread within the transaction block.

Table methods and table delete actions

If you want to perform cascade deletes or validations, it is always better to use delete actions than to add specific logic to your code.

Delete actions rely on relationships that are defined in metadata, so if you create a delete action, make sure that there is a relationship defined between the related tables. In Microsoft Dynamics AX, it is possible to specify the relationship that should be used for the delete action.

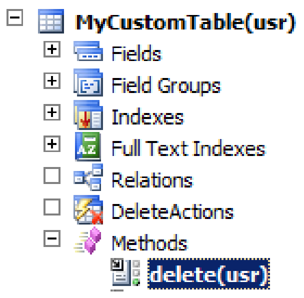


Figure 6 Delete action

Example scenario

In this example, if a record is deleted in a table, we want to delete records in a related table.

Bad solution: A related record is deleted by using code

The following example demonstrates how to do a cascade delete from code.

```
// MyCustomTable.delete
public void delete()
{
    MyCustomTableLog tableLog;
    ttsBegin;
    super();
    select forUpdate tableLog where tableLog.MyCustomTable == this.RecId;
    if (tableLog)
    {
        tableLog.delete();
    }

    ttsCommit;
}
```

Good solution: A related record is deleted by using the delete action

The following example demonstrates how to create a cascade delete by using the delete action.

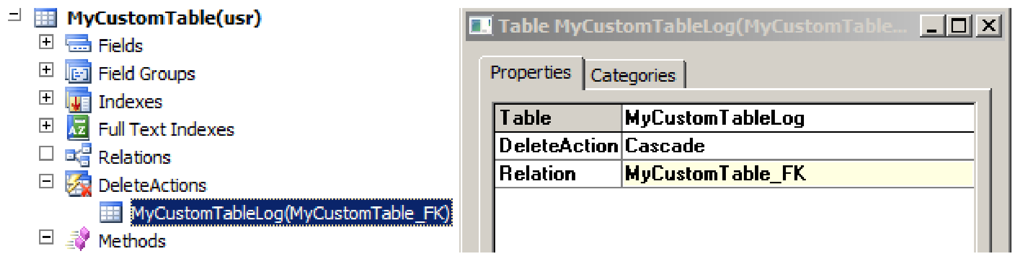


Table	MyCustomTableLog
DeleteAction	Cascade
Relation	MyCustomTable_FK

Figure 7 Creating a cascade delete

If you use **delete_from** to perform a set-based delete, and the **skipDeleteActions** method is called, you should add code manually to do the cascade delete, so that you have both performance and set-based behavior.

Example scenario

In this example, data needs to be deleted by using a set-based operation (from the parent table).

Bad solution: Data is deleted by using delete_from, but related table records are not deleted

```
void cleanData()
{
    MyCustomTable       customTable;
    MyCustomTableLog    tableLog;

    customTable.skipDatabaseLog(true);
    customTable.skipDataMethods(true);
    customTable.skipDeleteActions(true);    customTable.skipAosValidation(true);

    ttsbegin;

    delete_from customTable;

    ttsCommit;
}
```

Good solution: Data is deleted by using delete_from, and related table records are also deleted

```
void cleanData()
{
    MyCustomTable       customTable;
    MyCustomTableLog    tableLog;

    customTable.skipDatabaseLog(true);
    customTable.skipDataMethods(true);
    customTable.skipDeleteActions(true);    customTable.skipAosValidation(true);

    tableLog.skipDatabaseLog(true);
    tableLog.skipDataMethods(true);
```

```
        tableLog.skipDeleteActions(true);
        tableLog.skipAosValidation(true);

        ttsbegin;

        delete_from tableLog
            exists join customTable
                where customTable.RecId == tableLog.MyCustomTable;

        delete_from customTable;

        ttsCommit;
    }
```

Index considerations

An index is a physical database object that provides efficient access to data in the rows of a table based on key values.

It is important to understand the types of indexes that are available in SQL Server and Microsoft Dynamics AX, because indexing inappropriate columns, or not indexing appropriate columns, can significantly affect query performance.

Indexes are created on a subset of columns of a table and organized as B-trees. The top node of the B-tree is called the root node. The bottom level is called the leaf layer. Index levels between the root node and the leaf nodes are known as intermediate levels. Each index row contains a key value and a pointer to either an intermediate level page in the B-tree or an entry in the leaf layer of the index. The pages in each level of the index are linked in a double-linked list.

Indexes may be unique or non-unique. The key value of an index refers to the columns on which the index is defined. The key value of a non-unique index may not be unique.

SQL Server provides two main types of indexes:

- **Clustered index** – The leaf layer contains the data pages of the underlying table. The clustered index and the table share the same physical allocation. A column defined in the clustered index occupies the same physical space as the column in the table. The columns of the table that are not explicitly declared as part of the clustered index are stored physically adjacent to the clustered index columns (except for large object [LOB] columns). The data rows of the table are stored in the physical order of the clustered index columns. There can be no more than one clustered index per table. A table with no clustered index is called a heap.

  Indexes are designed to make frequent queries perform more efficiently. When the clustered index is used to access rows in a table, the leaf layer and the table data are stored together. After the leaf node for a row has been read, the entire row has been read (except for LOB columns). By contrast, a nonclustered index is stored separately from the data row of the table. When a nonclustered index is used to access rows in a table, the data rows must be read in separate I/O operations from the nonclustered index (unless the query is covered by the nonclustered index).

  Note: A clustered index can also be defined on a database view. This is called an indexed view in SQL Server. Indexed views are not currently supported in Microsoft Dynamics AX.

- **Nonclustered index** – The leaf layer is comprised of index pages rather than data pages. The clustered index key is used as a row locator for each nonclustered index entry.

  All indexes for a table are nonclustered indexes in SQL Server, except for the single clustered index.

Indexes (both clustered and nonclustered) can support the following operations:

- Joins
- Filters
- Sorts
- Aggregations (group by)

Nonclustered indexes are stored separately from each other and from the clustered index. Each nonclustered index occupies its own set of index pages.

Write-time I/O operations on nonclustered indexes include B-tree top, intermediate, and leaf levels. For example, in a nonclustered index with a single intermediate B-tree level, three index pages may be updated during a write operation for a single index entry: one for the top layer, one for the intermediate level, and one for the leaf level. More pages may be written if page splits occur at any of these levels.

By comparison, for each clustered index defined on a table, additional write I/O is required for the following scenarios:

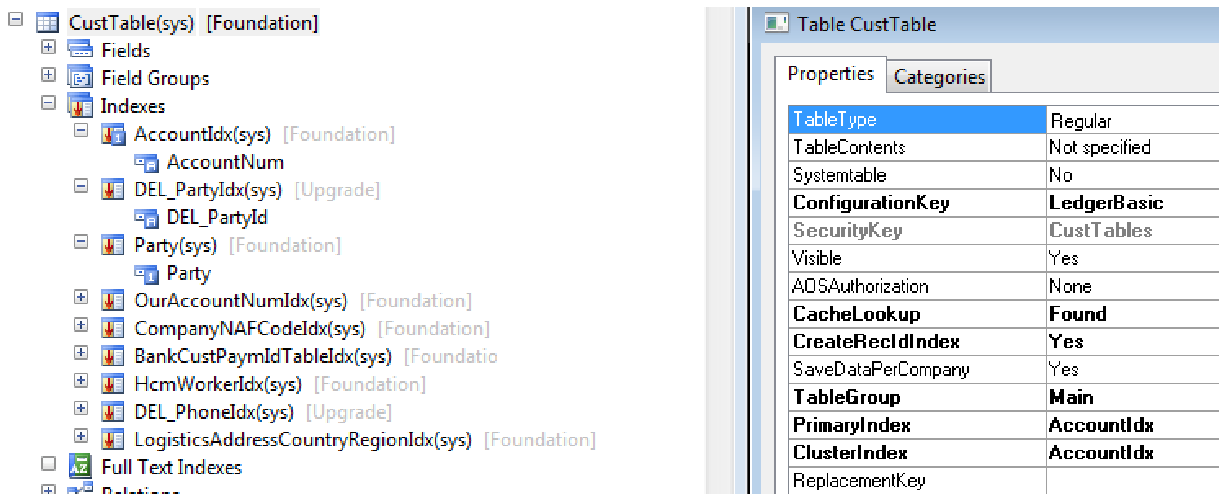
- Rows are inserted and deleted.
- Indexed columns are updated.

For more information, see SQL Server Books Online.

You can create an index in Microsoft Dynamics AX in the Application Object Tree (AOT) **Tables** node. Before you create an index, you must design the index based on the following:

- The columns to index
- The type of index
- The number of columns in the index
- The queries that you will use

The following screenshot illustrates the **Indexes** node for a table and the values that are set for **PrimaryIndex** and **ClusterIndex**.



```
CustTable(sys)  [Foundation]
  Fields
  Field Groups
  Indexes
    AccountIdx(sys)  [Foundation]
      AccountNum
    DEL_PartyIdx(sys)  [Upgrade]
      DEL_PartyId
    Party(sys)  [Foundation]
      Party
    OurAccountNumIdx(sys)  [Foundation]
    CompanyNAFCodeIdx(sys)  [Foundation]
    BankCustPaymIdTableIdx(sys)  [Foundatio
    HcmWorkerIdx(sys)  [Foundation]
    DEL_PhoneIdx(sys)  [Upgrade]
    LogisticsAddressCountryRegionIdx(sys)  [Foundation]
  Full Text Indexes
```

Table CustTable

TableType	Regular
TableContents	Not specified
Systemtable	No
ConfigurationKey	**LedgerBasic**
SecurityKey	CustTables
Visible	Yes
AOSAuthorization	None
CacheLookup	**Found**
CreateRecIdIndex	**Yes**
SaveDataPerCompany	Yes
TableGroup	**Main**
PrimaryIndex	**AccountIdx**
ClusterIndex	**AccountIdx**
ReplacementKey	

Figure 8 Indexes node, and PrimaryIndex and ClusterIndex table properties

Relationship of clustered and nonclustered indexes

A nonclustered index uses the clustered index key as a row locator if a clustered index is defined on the table. This means that all columns of the clustered index key are present in the nonclustered index entry, even though they may not be explicitly defined as part of the nonclustered index. A column on the clustered index is propagated in the nonclustered index only once, even if it is explicitly defined on the clustered index.

The following table illustrates the indexes on the AccountingEvent table.

Index name	Index description	Index keys
I_7456IDENTIFICATION	Nonclustered, located on PRIMARY	ACCOUNTINGDATE, TYPE, SOURCEDOCUMENTHEADER
I_7456RECID	Clustered, unique, primary key located on PRIMARY	RECID
I_7456SOURCEDOCUMENTHEADERIDX	Nonclustered, located on PRIMARY	SOURCEDOCUMENTHEADER

The clustered index is defined on the RecId column. Both of the nonclustered indexes include the RecId column, even though it is not explicitly defined as an index key. For example, the following query is covered by the index on SourceDocumentHeader (I_7456SOURCEDOCUMENTHEADERIDX):

```
SELECT SourceDocumentHeader, RecId
FROM AccountingEvent
```

The clustered index and table itself are not accessed when the query is run; only the nonclustered index is read. When the clustered index columns are not explicitly defined as part of the nonclustered index, they behave as included columns.

The following table describes three of the indexes defined on the VendTrans table.

Index name	Index description	Index keys
I_506ACCOUNTDATE1099IDX	Nonclustered, located on PRIMARY	DATAAREAID, ACCOUNTNUM, TAX1099FIELDS, TRANSDATE
I_506ACCOUNTDATEIDX	Clustered, located on PRIMARY	DATAAREAID, ACCOUNTNUM, TRANSDATE
I_506VOUCHERDATEIDX	Nonclustered, located on PRIMARY	DATAAREAID, VOUCHER, TRANSDATE

The clustered index (I_506ACCOUNTDATEIDX) is defined on three columns (DataAreaId, AccountNum, and TransDate). The nonclustered index I_506ACCOUNTDATE1099IDX is explicitly defined by using all three of the columns of the clustered index, in addition to TAX1099Fields column, so no columns of the clustered index are implicitly part of this nonclustered index.

The other nonclustered index, I_506VOUCHERDATEIDX, is defined on a subset of the columns in the clustered index. The clustered index columns that are not explicitly defined in this nonclustered index are still present, however.

The fact that all clustered index columns are present in nonclustered indexes on the same table is important for several reasons:

- The length of the clustered index key affects the length of the nonclustered index key.

  The length of nonclustered index keys influences how many nonclustered index entries are present on an index page. This influences how many index pages must be read when multiple index entries are read. A page (8 KB) represents the basic size of an I/O operation. The number of index entries per page influences the amount of I/O required to read multiple index entries, and I/O is a critical determinant of query performance.

Wider nonclustered index entries may require more write I/O during row inserts and when the indexed columns are updated.

- The presence of the clustered index key columns in the nonclustered index affects queries that the nonclustered index can cover.

This is true even when the clustered key columns are not explicitly defined on the nonclustered index. Queries that are covered by nonclustered indexes incur less I/O than queries that require access to the clustered index and table. Nonclustered indexes tend to be much narrower than clustered indexes. This is because the clustered index keys are on the data row pages. In a multi-row query, the amount of I/O consumed for a query that is covered by a nonclustered index can be less than a similar query that is either covered by a clustered index or not covered by an index.

Nonclustered index coverage trades off against index entry width.

Best practices for clustered indexes

A clustered index is required on all permanent Microsoft Dynamics AX tables.

Clustered indexes are often defined on the primary key (PK) of a table. The PK usually consists of the single RecId column, so the default practice is to define the clustered index on RecId. However, this is not necessarily an optimal indexing strategy and may need to be adjusted.

The Microsoft Dynamics AX TableGroup, EntityRelationshipType, supertype/subtype, and date effective classifications can be used as a starting point to determine the optimal type of clustered index for a table.

Relationship types are often called "association" or "intersection" tables. They resolve many-to-many (m:n) relationships. These tables are characterized by having two or more FK columns comprising an alternate key (AK). Relationship Type is enumerated as EntityRelationshipType "Relationship" in the AOT; it is not part of the TableGroup taxonomy.

Supertype/Subtype (or SCsc) tables participate in type hierarchies and have the AOT property **SupportInheritance**=**Yes**. Tables that are subtypes also have the property **Extends** set to their supertype table.

Table property to assess	What to index	Examples
Framework table group	RecId or PK.	SysAOSMeasurement
Reference table group	**Reference** tables are generally referenced by FKs in other tables, such as **Main** and **Transaction** tables. The clustered index is defined on the key value most frequently used as an FK on tables that reference the **Group** table – usually RecId.	LogisticsLocationRole, DirNameSequence
Parameter table group	RecId or PK.	CustParameters, VendParameters
Group table group	**Group** tables are generally referenced by FKs in other tables, such as **Main** and **Transaction** tables. The clustered index is defined on the key value most frequently used as an FK on tables that reference the **Group** table – usually RecId.	CustGroup, VendGroup
Main table group	The clustered index is generally defined on the key value most frequently used as an FK on tables that reference the **Main** table – usually the PK.	CustTable, VendTable

Table property to assess	What to index	Examples
Transaction table group	The clustered index should generally be defined on the columns most frequently used to query the table. Performance assessments in the lab and the field have shown overall better Microsoft Dynamics AX performance compared to a scenario in which Transaction table clustered indexes are defined on RecId, even though RecId may reduce the frequency of page splits during inserts.	CustTrans, VendTrans
WorksheetHeader table group	In most cases, the clustered index is created on the key value most frequently used as the FK on tables that reference the **WorksheetHeader** table – usually the PK. However, a clustered index of RecId is not optimal for range queries that are used, for example, in list page forms; but because these ranges can often be sorted and filtered by user-selected criteria, there may be no single best clustered index key value for range query support.	SalesTable
WorksheetLine table group	The clustered index is generally defined on the AK that comprises the FK to the corresponding **WorksheetHeader** table and the attribute that sequences the lines. These tables tend to be accessed most frequently by range queries on this AK. The FK of the **WorksheetHeader** table should be defined on the left of the other column in the clustered index.	SalesLine
TransactionHeader table group		GeneralLedgerEntry
TransactionLine table group		GeneralLedgerAccountEntry
Relationship Type	The clustered index is generally defined on the FK columns that comprise the AK. The columns most frequently used in queries should occur leftmost in the index. For example, in DirPartyLocation, the AK is comprised of the FKs of DirParty and LogisticsLocation FKs, and also the ValidFrom date. Most queries access by Party (for example, find all the locations for a given party) rather than by location (for example, find all the parties for a given location); therefore, the FK to DirParty should occur as the leftmost column in the clustered index.	DirPartyLocation
Supertype/Subtype	Use RecId for a table that is a subtype of another table. Generally, use RecId for tables that are supertypes and not subtypes.	
Date-effective	Follow the same guidelines for Relationship Type; most date-effective tables are Relationship Types. Define the clustered index on the AK that includes ValidFrom.	

Use the TableGroup taxonomy listed earlier as a starting point.

Do not use RecId automatically, because in many cases, it is not the optimal clustered index.

Do not use columns that are subject to updates. When a column in a clustered index is updated, the row may have to be moved to a new page, and all nonclustered index entries for that row will have to be updated. This increases the I/O cost of updates.

Clustered indexes do not necessarily have to be unique. When a clustered index is non-unique, SQL Server adds a 4-byte uniquifier integer to the index entry. This only happens when a duplicate entry is detected; otherwise, the uniquifier is NULL and consumes no space. If there are few duplicate entries in the clustered index, the incremental cost of a non-unique index is low. Do not add a column to a clustered index solely to make it unique.

Consider the size of the clustered index key.

- Microsoft Dynamics AX indexing guidelines do not impose a fixed maximum size to a clustered index key. However, because the clustered index key is used as a row locator in nonclustered indexes on the same table, a long clustered index key can increase the size of nonclustered index keys.
- All clustered index columns occur in nonclustered index entries (of the same table), regardless of whether they are explicitly defined on the nonclustered index. (For more details, see the section about nonclustered indexes that follows.)

Best practices for nonclustered indexes

The number of nonclustered indexes on a table can have a significant effect on the performance of writes to that table. There is a trade-off between write I/O and read I/O. A higher number of nonclustered indexes on a table can improve read performance by avoiding index scans in favor of seeks and, in some cases, by covering queries. These benefits depend on correct index design. Poorly designed indexes can reduce performance.

Queries that result in lookups by AKs can read the kernel record cache and reduce database reads, whereas those that do not include an AK cannot. Do not add extra key fields to an AK if they are not part of the AK. Adding extra key fields will reduce the ability to use the cache.

Make indexes into AKs if they are unique.

General indexing best practices

The primary index and clustered index are critical properties for a table. You should analyze and set the properties correctly.

You should analyze the usage and queries for customized tables (new and modified) and create necessary indexes. Depending on how the table is queried, necessary indexes should be created. Indexes can help boost performance for queries, but at the same time, they add cost to inserts and updates. Analyze your usage scenarios before creating indexes.

Design indexes to avoid index scans. An index scan requires the entire index to be read. A scan of the clustered index is equivalent to a table scan.

Use nonkey or included columns in nonclustered indexes to provide query coverage. Benefits include the following:

- They impose somewhat less overhead than key columns. They are stored only at the leaf level of the index. Nonkey columns on nonclustered indexes are similar to columns in a table that are not part of the clustered index key.

  A common example is in date-effective tables, where the ValidTo column is defined as an included column.
- The main performance trade-offs of nonkey columns are that fewer index rows may fit on a page, and there is increased I/O in write operations. As with key columns, avoid including unnecessary nonkey columns in indexes.

  Important: SQL Server Database Tuning Advisor (DTA) often recommends indexes with a large number of nonkey columns. If you are using DTA, make sure that changes are validated by a database administrator (DBA) and tested thoroughly before implementing its recommendations, especially recommendations for indexes with a lot of nonkey columns.

- In general, if you know that there are specific queries that benefit from the presence of the clustered key columns in the nonclustered index, you can explicitly define them as nonkey (included) columns to ensure that they will continue to cover even if the clustered index is changed. However, this should not be done as a matter of course but only when there is a specific need.

  For example, consider the AccountingEvent table indexes. Assume that your system includes a frequently executed query that was covered because RecId was effectively included in the index. If the clustered index on this table was changed at a later time so that RecId was no longer a clustered index key, the nonclustered index would no longer cover the query. In this case, you can explicitly define RecId as a nonkey column on the nonclustered index. When the clustered index key is comprised of RecId, this definitional change to the nonclustered index has no real effect in the physical storage of the index. However, by explicitly making RecId a nonkey column, you preserve the covering effect of the index if the clustered index is changed later.

- Key information about nonkey columns includes the following:
  - A column cannot be defined as both a key and nonkey (included) column in the same index.
  - The same column cannot be defined as a nonkey column more than once.
  - A computed column can be used as a nonkey column.
  - Included columns can be of data types that cannot be used as key columns, such as NVARCHAR(MAX) and XML. Included columns are not considered by the database engine when it calculates the number of index key columns or index key size (16 columns and 900 bytes, respectively); a maximum of 1,023 nonkey columns can be included in and index, and the total size of the nonkey columns is limited by the size of the columns.

Ensure that the index column order is appropriate. The leftmost column of an index must be used in a query for a seek operation rather than a scan operation to be performed, or in some cases, for the index to be used at all.

Index date-effective tables appropriately. Date-effective tables are modeled with an AK that includes the ValidFrom column. In some models, the ValidTo column may also be included in the AK, but this is not necessary for uniqueness, and it should be removed from the AK constraint. If the ValidFrom column is a key column of the clustered index, the ValidTo column should not also be a key column of the clustered index. If the ValidFrom column is a key column of a nonclustered index, the ValidTo column should be made a nonkey (included) column in the nonclustered index, which provides coverage for range queries that involve both ValidTo and ValidFrom columns.

Measure index performance. A simple way to measure the improvement due to an index change is to capture one or more SQL statements that lead to the index analysis and then "replay" the SQL statements after the index change is in place. SQL Server Profiler and SQL Server Management Studio are the best tools to accomplish this.

Note: For any performance testing to be effective, data volumes must be sufficiently realistic; otherwise, the execution plans produced by the following procedure will not be useful for determining actual performance.

1. Use Profiler to prepare the application for tracing by navigating to the point where you want to capture SQL activity. This will help minimize the volume of SQL statements captured by Profiler.
2. Prepare Profiler. Under the **Stored Procedures** event class, select the **RPC Starting** event. Then start Profiler.
3. Start the application process, and let it run to completion.

4. When the application process is completed, stop Profiler, and save the trace data by using a meaningful trace file name. Open the trace file, and search for one or more SQL statements that you expect to take advantage of the proposed index changes. The SQL statements will have the following form:

   ```
   declare @p1 int
   set @p1=NULL
   declare @p2 int
   set @p2=0
   declare @p5 int
   set @p5=28688
   declare @p6 int
   set @p6=8193
   declare @p7 int
   set @p7=2
   exec sp_cursorprepexec @p1 output,@p2 output,N'@P1 nvarchar(5),@P2 nvarchar(21)',N'SELECT
   A.ACCOUNTNUM,A.RECVERSION,A.RECID FROM LEDGERJOURNALTRANS A WHERE ((DATAAREAID=@P1) AND
   (ACCOUNTNUM=@P2))',@p5 output,@p6 output,@p7 output,N'ceu',N'130400'
   select @p1, @p2, @p5, @p6, @p7
   ```

5. After the index changes have been made, the SQL statements can be executed by using SQL Server Management Studio. You can enable viewing of the execution plan for the SQL statements from **Query** > **Include Actual Execution Plan** or by pressing Ctrl+M. Save the execution plan for future reference by right-clicking it and then selecting **Save Execution Plan As**.

6. Precede the SQL statement with the following SQL command:

   ```
   DBCC FREEPROCCACHE
   GO
   ```

   Then execute the SQL statement.

7. It is important to verify that the proposed index changes are effective across a range of data; therefore, the SQL statements should be executed with multiple parameter replacement values representing typical, low, and high distribution values for the parameterized columns in the statement's WHERE clause.

   In the preceding statement, the parameterized WHERE clause columns are ((DATAAREAID=@P1) AND (ACCOUNTNUM=@P2)); the parameter replacement values are N'ceu',N'130400'.

   To produce a list of test replacement values for the preceding SQL, you can construct a query to return row counts by the combinations of the WHERE clause columns, as follows:

   ```
   SELECT DATAAREAID, ACCOUNTNUM, COUNT(*)
   FROM LEDGERJOURNALTRANS
   GROUP BY DATAAREAID, ACCOUNTNUM
   ORDER BY COUNT(*) DESC
   ```

   The list produced by this query provides candidate replacement values that can be supplied in place of N'ceu' and N'130400' in the sample SQL statement. Select the first (high) and last (low) values, and execute the SQL statement with these as parameter replacement values. Observe any changes in the execution plan.

Index design best practices

Index design is not an exact science. Often, it is necessary to make some assumptions about how the application will behave, particularly with regard to read and write rates for critical data, and where performance trade-offs will be most beneficial to most users.

Prerequisites

- Understand the SQL Server index storage architecture.
- Understand how indexes are used in queries.
- Understand how data is most likely to be accessed.

Initial index design

Identify all indexes in the physical data model (PDM).

Microsoft Dynamics AX modeling guidelines require all indexes to be identified in the PDM. We recommend that you use Microsoft Visio for this process.

In Visio, unique indexes are identified by the AK (alternate key) designation; non-unique indexes are identified by the IE (inversion entry) designation. The order in which columns occur in a composite index is important and must be accurately reflected in the PDM.

Visio does not currently support the following:

- Distinguishing clustered and nonclustered indexes
- Identifying nonkey (included) index columns

Queries for reuse

If you analyze your requirements and scenarios with the complete picture of your business needs in mind, you can design your queries before you design the actual business process or UI components.

Designing queries before business processes can help you use the same query for a business process, a form, and a report. For example, a business process might require a query of open sales order documents. You could reuse that query in a form to show a list of open sales orders.

Developers often try to solve the problem for only one component – for example, by building a query for a single business process, and then using traditional form data sources for forms or creating new queries for reports.

Business processes

This section outlines best practices for business processes.

Using events

We expect that all implementations of Microsoft Dynamics AX will be customized, because customers differ in the way they do their day-to-day business. It is not possible to have one application that meets the requirements of all customers. The base version of Microsoft Dynamics AX acts as a foundation that customers and partners customize. Customers invest time and money in customizations. At the same time, Microsoft works to improve the foundation by introducing new features and improving existing ones. For customers to benefit from new features, they must upgrade their customized code to later Microsoft Dynamics AX versions.

A problem with the customization model prior to Microsoft Dynamics AX 2012 was that customizations could be broken when a new version was released, if Microsoft restructured code in a lower layer. Fixing earlier customizations during upgrade can be expensive.

The use of events has the potential to lower the cost of creating and upgrading customizations. Developers who create code that is often customized by others should create events in places where customizations typically occur. Then developers who customize the original functionality in another layer can subscribe to an event. When the customized functionality is tied to an event, the underlying application code can then be rewritten and will have little impact on the customization

Example scenario

In this example, a business requirement is to support copying from another document, similar to what is available for purchase orders. Purchase orders have functionality in which in order lines can be copied from another purchase order's lines, or from other documents such as confirmations, product receipts, and invoices.

Bad solution: Customizing a base layer class

The following example customizes a base layer class. The code depends on how the base class is structured.

If the base class is redesigned, you will have to rewrite your customizations. If, in a new Microsoft Dynamics AX version, Microsoft removes that class, creates a new class called **PurchCopyService**, and completely changes the structure of the code inside that class, a developer must first understand the new design and then rewrite his or her code at the correct place in the new class. This is expensive and time consuming.

\Classes\PurchCopying\purchCopyingCopyLineDefault

```
protected void copyLine(TmpFrmVirtual _tmpFrmVirtualLines)
 {
    ..
..
..
..
..

    switch(_tmpFrmVirtualLines.TableNum)
    {
        /* New code block which usually developer would add here.*/
        case tableNum(MyCustomizedDocument):
        ..
..
            break;
        /* End of New code block*/

        case tableNum(PurchLine):
        ..
..
            break;

        case tableNum(VendInvoiceTrans):
        ..
..
            break;

        case tableNum(VendPackingSlipTrans):
        ..
..
            break;

        case tableNum(PurchLineHistory):
        ..
..
            break;
        default:

            // the code under default is only for customizations in the higher layers.
            result = new Struct('str currencyCode; common fromTrans');
            this.purchCopyingCopyLineDefault(this, purchLine, _tmpFrmVirtualLines, result);
            currencyCode = result.value('currencyCode');
            fromTrans = result.value('fromTrans');
            break;

        ..
..
        // Call the copyLineFinal delegate -- this part of the code for higher layer
customizations only.
        this.purchCopyingCopyLineFinal(this, purchLine, _tmpFrmVirtualLines);
    }

}
```

Good solution: Customization by using events

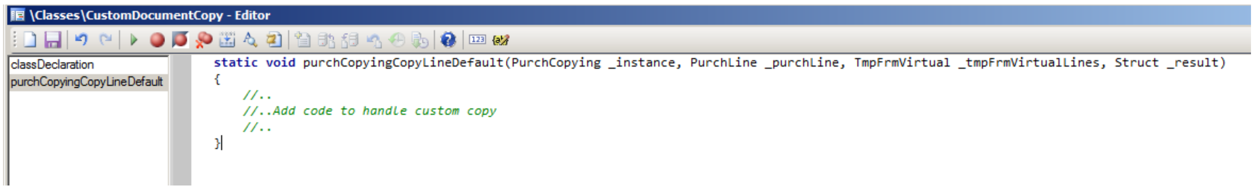
If we use events in this case, even if the base layer class is redesigned, there would be minimal effort required from a developer.

\Classes\PurchCopying\purchCopyingCopyLineDefault

```
protected void copyLine(TmpFrmVirtual _tmpFrmVirtualLines)
{
    ..
..
..
..
..

    switch(_tmpFrmVirtualLines.TableNum)
    {

        case tableNum(PurchLine):
        ..
..
            break;

        case tableNum(VendInvoiceTrans):
        ..
..
            break;

        case tableNum(VendPackingSlipTrans):
        ..
..
            break;

        case tableNum(PurchLineHistory):
        ..
..
```

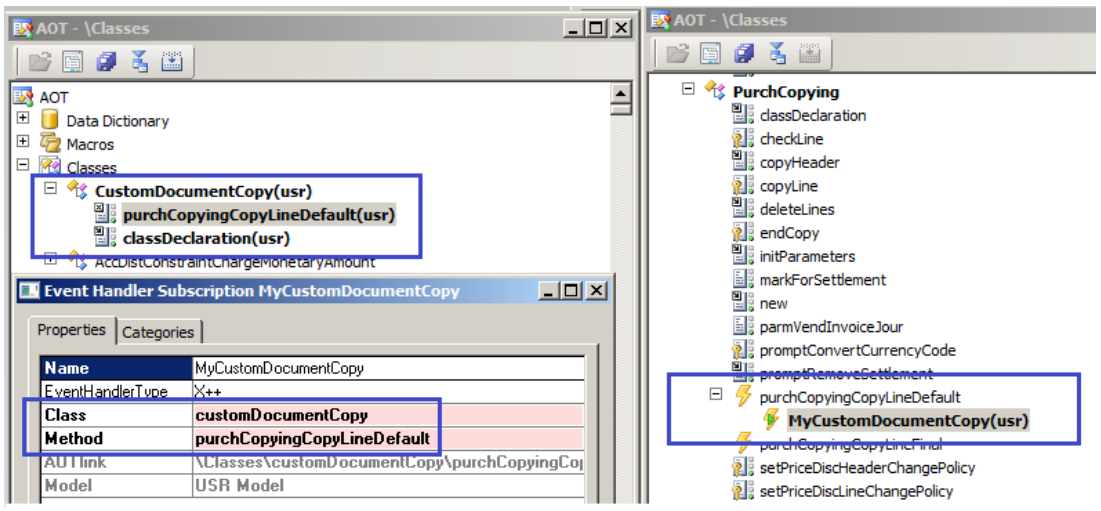
```
                break;

            default:

                // the code under default is only for customizations in the higher layers.
                result = new Struct('str currencyCode; common fromTrans');
               /* Delegate called by base layer. */
                this.purchCopyingCopyLineDefault(this, purchLine, _tmpFrmVirtualLines, result);
                currencyCode = result.value('currencyCode');
                fromTrans = result.value('fromTrans');
                break;

            ..
    ..
            // Call the copyLineFinal delegate -- this part of the code for higher layer
    customizations only.
            this.purchCopyingCopyLineFinal(this, purchLine, _tmpFrmVirtualLines);
        }

    }

    delegate void purchCopyingCopyLineDefault(PurchCopying _instance, PurchLine _purchLine,
    TmpFrmVirtual _tmpFrmVirtualLines, Struct _result)
    {
    }
```

The base layer is calling the delegate purchCopyingCopyLineDefault. We can now add a new event handler to handle copying from our custom document.


\Classes\CustomDocumentCopy - Editor
classDeclaration
purchCopyingCopyLineDefault

```
static void purchCopyingCopyLineDefault(PurchCopying _instance, PurchLine _purchLine, TmpFrmVirtual _tmpFrmVirtualLines, Struct _result)
{
    //..
    //..Add code to handle custom copy
    //..
}
```

Figure 9 Code to handle copying


AOT - \Classes

AOT
Data Dictionary
Macros
Classes
CustomDocumentCopy(usr)
purchCopyingCopyLineDefault(usr)
classDeclaration(usr)
AccDistConstraintChargeMonetaryAmount

Event Handler Subscription MyCustomDocumentCopy

Properties | Categories

Name	MyCustomDocumentCopy
EventHandlerType	X++
Class	customDocumentCopy
Method	purchCopyingCopyLineDefault
AOTlink	\Classes\customDocumentCopy\purchCopyingCo
Model	USR Model

AOT - \Classes

PurchCopying
classDeclaration
checkLine
copyHeader
copyLine
deleteLines
endCopy
initParameters
markForSettlement
new
parmVendInvoiceJour
promptConvertCurrencyCode
promptRemoveSettlement
purchCopyingCopyLineDefault
MyCustomDocumentCopy(usr)
purchCopyingCopyLineFinal
setPriceDiscHeaderChangePolicy
setPriceDiscLineChangePolicy

Figure 10 Adding an event handler

In this case, when the base layer class is removed, and a new class (**PurchCopyService**) is created, the same delegate would be created in the base layer. To upgrade to new version, we would only have to subscribe our event handler to the new delegate.

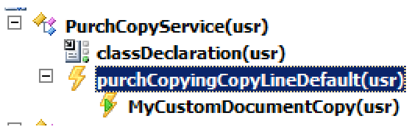

PurchCopyService(usr)
classDeclaration(usr)
purchCopyingCopyLineDefault(usr)
MyCustomDocumentCopy(usr)

Figure 11 Adding the event handler to the new class

In this example, the class was removed from the base layer; however, it may be that the method was changed heavily in the base layer, the same class was restructured, or the method was removed. In all such cases, previous delegates would be preserved, and if you have your customized event handlers subscribed correctly, you would not need any changes in code.

Note: If you feel that the base layer shipped by Microsoft should have specific additional delegates, feel free to provide feedback to daxcode@microsoft.com.

Pre-handlers or post-handlers

Events can also be used if customization is done at the start and end of a method.

Customers and partners often customize methods in classes. If you know that the intent of the customized code is just to be executed before or after the base layer method, you can use pre-event or post-event handlers. Many times, when we add code, it does not matter whether it is placed in a base code block, or before or after it.

Example scenario

Customize existing logic to determine whether depreciation is finished, check additional conditions, and then return true or false.

Bad solution: Modify a base method

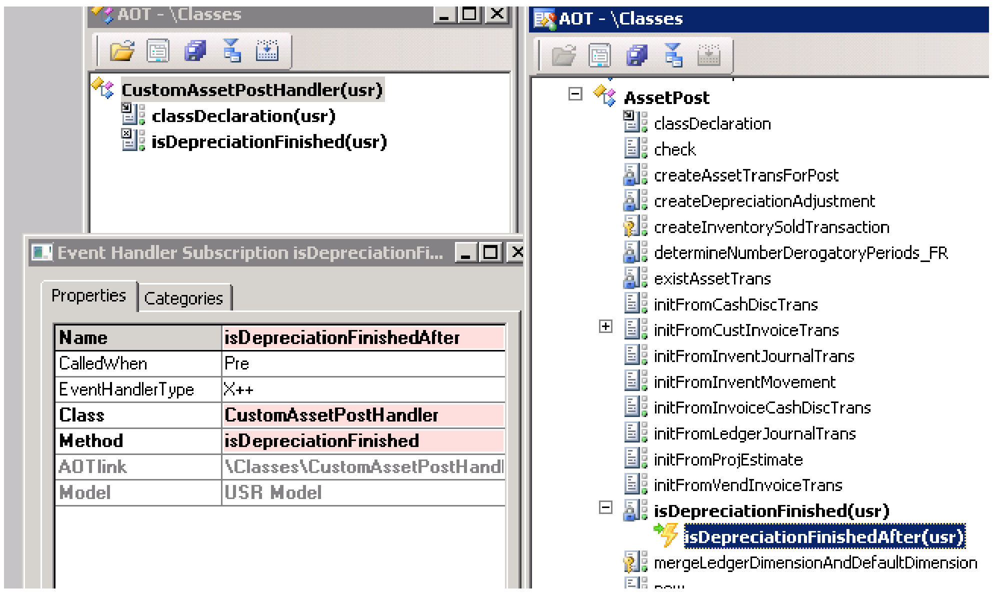
With this approach of modifying base layer code, we have added customized code into a base layer method. If Microsoft changes code in the base layer in a future version, this would come up in upgrade as a conflict, which would require manual efforts to fix.

The **\Classes\AssetPost\isDepreciationFinished** method is used to check whether depreciation is completed.

```
private boolean isDepreciationFinished(AssetBook _assetBook)
{
    boolean retValue;

    if (assetTrans.AmountMST < 0
        && AssetSumCalc_Trans::newAssetYear(
            _assetBook.AssetId,
            _assetBook.BookId).netBookValue() <= _assetBook.ScrapValue)
    {
        retValue = true;
    }
    /* Start of custom code*/
    if (retValue)
    {
        retValue = yearDiff(_assetBook.AcquisitionDate,systemDateGet()) > 10;

    }
    /* End of custom code*/
    return retValue;
}
```

Good solution: Create a post-event handler

In this example, we create a post-event handler. This way, even if the base layer code is modified in the future, the customized code in the event handler will not be affected, and upgrade will not be an issue.



AOT - \Classes
CustomAssetPostHandler(usr)
classDeclaration(usr)
isDepreciationFinished(usr)
Event Handler Subscription isDepreciationFi...
Properties
Categories
AOT - \Classes
AssetPost
classDeclaration
check
createAssetTransForPost
createDepreciationAdjustment
createInventorySoldTransaction
determineNumberDerogatoryPeriods_FR
existAssetTrans
initFromCashDiscTrans
initFromCustInvoiceTrans
initFromInventJournalTrans
initFromInventMovement
initFromInvoiceCashDiscTrans
initFromLedgerJournalTrans
initFromProjEstimate
initFromVendInvoiceTrans
isDepreciationFinished(usr)
isDepreciationFinishedAfter(usr)
mergeLedgerDimensionAndDefaultDimension

Name	isDepreciationFinishedAfter
CalledWhen	Pre
EventHandlerType	X++
Class	CustomAssetPostHandler
Method	isDepreciationFinished
AOTlink	\Classes\CustomAssetPostHandl
Model	USR Model

Figure 12 IsDepreciationFinished post-event handler

\Classes\CustomAssetPostHandler\isDepreciationFinished

```
static void isDepreciationFinished(XppPrePostArgs ppArgs)
{
    boolean         retValue;
    AssetBook       assetBook;

    retValue = ppArgs.getReturnValue();
    assetBook = ppArgs.getArg(identifierStr(_assetBook));

    if (retValue)
    {
        retValue = yearDiff(assetBook.AcquisitionDate,systemDateGet()) > 10;
    }

    ppArgs.setReturnValue(retValue);
}
```

Event best practices

The following section summarizes dos and don’ts to consider when working with events.

Do consider

For customizations, subscribe to the already-defined events in the base layer.

Favor defining events over using pre-events and post-events on methods. Events are strong interfaces, whereas methods are an implementation artifact. The uses and parameters of methods can change over time, or they may disappear altogether.

Don’t consider

Do not try to raise an event from outside the class in which the event is defined. By design, the X++ compiler disallows raising events from outside the class.

Do not make customizations that rely on the order in which event handlers are executed. No guarantees are made in the runtime environment about the order in which the event handlers are called.

Batch and asynchronous processing

For custom business processes, always evaluate whether they need to be executed synchronously or asynchronously.

- Synchronous execution is when the system completes a business process requested by a user before returning control back to the user.
- Asynchronous execution is when the system notes a business process requested by a user, returns control back to the user, and executes the process at that time or at a later time on an AOS instance, as scheduled.

Business processes that involve volume transactions and processing should always be executed asynchronously at a time of day when normal users are not working. Developers should always design processes to support Batch functionality, so that there is an option to run the process both synchronously and asynchronously.

An example is the invoicing of sales orders.

Parallel processing

When developers design business processes, they should divide them into multiple operations and determine which operations can be executed independently of the others. For those operations, multiple batch tasks should be used.

In cases of volume processing, in which groups of data can be divided and processed in isolation, you should create parallel batch jobs.

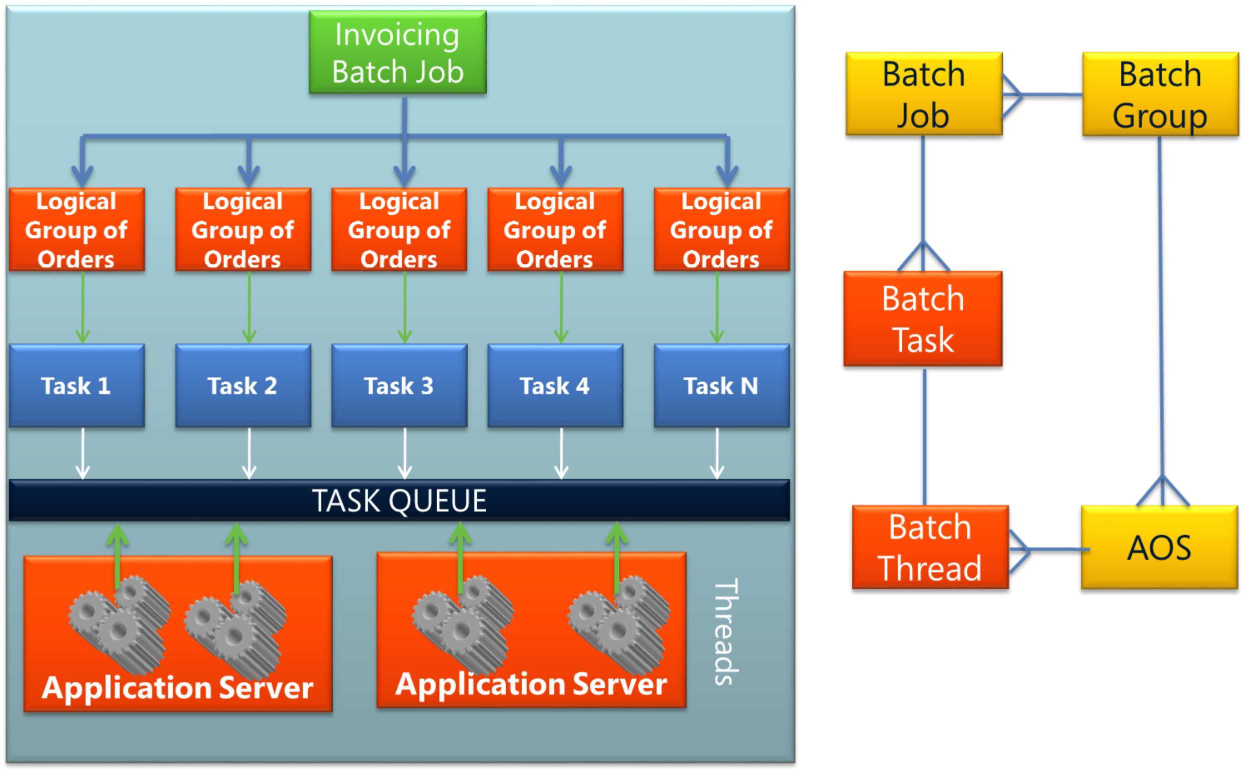


Figure 13 Processing batch jobs

For example, if there are 1 million sales orders, you can create multiple batch jobs on the basis of criteria such as customer, customer group, region, and so on.

Microsoft Dynamics AX 2012 and Microsoft Dynamics AX 2009 both have the ability to break down a batch job into small, manageable fragments and process them independently in parallel. This ability is critical to improving the throughput and response time for batch jobs, and to shrink the required batch processing window. The following are three common approaches to breaking a batch job into smaller fragments:

- **Batch bundling** – In this model, a static number of tasks is created. Split by grouping the work items together into bundles that are of approximately equal size, and then assigning each bundle to a task. Each worker thread will process a bundle of work items before picking up the next bundle. This approach is best if all of the batch tasks take roughly the same amount of time to process each bundle. In an ideal situation, each worker would be actively doing the same amount of work. This approach is less desirable in scenarios in which the workload is variable because of data composition or a difference in server hardware. In these situations, you may end up waiting for the last few threads of the largest bundle to be completed while other threads have already been completed.

- **Individual task modeling** – In this model, parallelism is achieved by creating a separate task for each work item. Tasks are created for each work item. Create a 1:1 mapping between each task and the work item. This eliminates the need for pre-allocation. Because each work item is independently handled by a worker thread, workload distribution will be more consistent. This approach eliminates the problem of a number of large work items getting bundled together and eventually dragging the batch response time.

  The disadvantage to this approach is that it if a large number of work items needs to be processed, a large number of batch tasks must be created. The overhead for the batch framework to maintain this number of tasks can be very high. The batch framework checks several conditions, dependencies, and constraints each time a set of tasks is completed and a new set of tasks is picked up for execution.
- **Top picking** – In this model a static number of tasks is created (similarly to bundling). No pre-allocation is done (similarly to individual task modeling). Because no pre-allocation is done, and the method does not rely on the batch framework to separate the work items, you must maintain a staging table that contains all of the work items. Maintaining this staging table to track the progress of the work items requires overhead, but it is much smaller than the overhead of the batch framework. After the staging table is populated, the worker threads can start processing by fetching the next available item from the staging table, and they can continue until there are no more work items left. Therefore, no worker threads will be idle while other worker threads are being overloaded. To implement this, we use the PESSIMISTICLOCK hint along with the READPAST hint. These hints enable worker threads to fetch the next available item without being blocked.

No framework addresses all issues. Bundling can create uneven distribution of the workload. Individual task modeling addresses uneven distribution of the workload, but when it is used for large numbers of items, it can cause significant batch framework overhead. Top picking also addresses the uneven distribution problem, but it can also have significant overhead if there are large numbers of tasks.

For detailed examples, see http://blogs.msdn.com/b/axperf/.

Services

Use services to code business processes. Microsoft Dynamics AX 2012 supports the following three kinds of services:

- Document services are query-based services that can be used to exchange data with external systems by sending and receiving XML documents. These documents represent business entities, such as customers, vendors, or sales orders.
- Custom services can be used by developers to expose any X++ logic, such as X++ classes and their members, through a service interface.
- System services are provided by Microsoft Dynamics AX. System services include the Query service, the Metadata service, and the User Session service. System services are not customizable, and they are not mapped to any query or X++ code.

Developers should use the existing services exposed in the Microsoft Dynamics AX base layer. Expose any new business processes through services.

SysOperation framework

Use the SysOperation framework (formerly known as the Business Operation framework, or BOF) when you extend Microsoft Dynamics AX by adding new functionality that may require batch processing. The SysOperation framework replaces the RunBase Framework and provides infrastructure for creating user interaction dialog boxes and integration with the batch server for batch processing.

Important features of the SysOperation framework include the following:

- It enables menu-driven execution or batch execution of services.
- It calls services in synchronous or asynchronous mode.

- It automatically creates a customizable UI based on the data contract.
- It encapsulates code to operate on the appropriate tier (prompting on the client tier, and business logic on the server tier).

Combining the SysOperation framework and services creates a good foundation for reusing business processes for multiple user interfaces. For example, you can use the sales order invoice service for both the rich client and Enterprise Portal for Microsoft Dynamics AX, or for a custom C# application.

The SysOperation framework supports a dynamic UI and different execution modes from X++, which makes development very clean and reusable.

For a comparison of the SysOperation and RunBase frameworks, and to view sample code that illustrates interactive and batch execution, see the white paper Introduction to the SysOperation Framework.

Unit of work

If you are dealing with entities that are normalized across multiple tables, such as Party, Location, Postal Address, and Product configuration, a unit of work should be used to manage database transactions. You can provide a series of specific individual rows to a UnitOfWork object, in the form of table buffer variables. The UnitOfWork object successfully processes each row, or it rejects all changes. The **UnitOfWork** class automatically determines the correct sequence of delete, insert, and then update operations on tables that are linked by a foreign key relationship.

Application framework

With Microsoft Dynamics AX 2012, major functionalities have changed. A huge amount of work was done to improve the application framework to support more functionality. The following are some of the key features implemented in Microsoft Dynamics AX 2012. To best take advantage of these new features, be sure to read the following references:

- Source document
- Global address book (GAB)
- Account and dimensions
- Product and dimensions

Coding best practices

This section provides guidance to help you avoid common pitfalls and prevent best practice violations.

Customizing code

If base layer code needs to be replicated or used at other places, it is always better to extend the existing classes and modify the derived class for the change in behavior, rather than creating completely new classes and then copying entire code from the base class.

Following this advice makes it easier to upgrade code: when base layer code is changed, it must be replicated again; however, if you have created an extension, only the modified code must be restructured.

Create classes and methods so that the same piece of code can be reused in multiple places. Avoid creating long methods. They make code difficult to read, hard to debug, and extremely difficult to upgrade and maintain.

Remove dead code to avoid upgrade and maintenance costs.

For versions before Microsoft Dynamics AX 2012, it is extremely important that you not add "BP deviations documented" to code unless you have validated that security is not being bypassed for display methods and other code paths.

Where to add custom code

Create customizations in the appropriate location. Create code for reuse as much as possible, but create it at the lowest appropriate location. For example, if anything is required only in a form, do not put it at the table level. The following examples describe where we recommend that you place the code:

- If it is related to the UI, place the code on the appropriate UI elements, or create classes to handle scenarios specific to the UI. For example, you can create classes that handle controls, number sequences in forms, dialog boxes, and so on.
- If it is related to a business process, place the code in classes.
- If it is directly related to tables and schemas, place code on the tables.

Consume existing Microsoft Dynamics AX classes and services instead of directly querying tables.

Become familiar with the base layer features.

Do not write custom code that duplicates base functionality.

Important: Directly updating or deleting data from Microsoft Dynamics AX tables is not supported.

Coding standards

The following list provides coding standards to follow:

- Favor the use of positive logic.
- Use constants instead of numeric literals.
- Use enumerations instead of constants.
- Use explicit access modifiers.
- Do not use method parameters as an l-value.
- Arguments passed in a method should not be modified, and only the value should be used.

Example scenario

In this example, parameters need to be passed to a method, and the values also need to be modified.

Bad solution: Parameters are passed and modified in the method

In this example, parameters passed to **methodxyz()** are modified in the method.

```
Void methodxyz(SalesId _salesId)
{
    Select * from salesTable where…;
    _salesId = salesTable.SalesId;
    ..
    …
    ..
}
```

- Do not mix data types in code.

Example scenario

In this example, fields of different types are required.

Bad solution: Fields are mixed in code

```
Table1.PartyId = EmplTable.EmplId;
```

Or

```
Table1.DocumentNum = any2str(SalesTable.RecId);
Table1.DocumentNum = custInvoiceJour.InvoiceId..
```

- Analyze the changes suggested by best practice checks, and review the intent of the best practice. For example, if hard-coded numbers are used, and you create a macro, it should have a meaningful name. Replacing 9 with #define.nine(9) is not good; a better option would be #mAXLimit.(9), #BlockSize.(9), or #defaultValue.(9).
- Use the cross-company keyword with X++ select statements to query across the current company. This is preferred to using **changeCompany()**.

Example scenario

In this example, records from multiple companies are fetched.

Bad solution: Cross-company by using the changeCompany() API

In this example, records from multiple companies are fetched, and **changeCompany()** is used to change the company.

```
static void processCust(Args _args)
{
    CustTable custTable;
    DataArea  dataArea;

    while select Id from dataArea
            where !dataArea.isVirtual
    {
        changeCompany(dataArea.id)
        {
            custTable = null;
            while select Party from custTable
            {
                //..
                //..
                //..
            }
        }
    }

}
```

Good solution: Cross-company by using the crosscompany option with select

In this example, records from multiple companies are fetched by using the **crosscompany** option with **select**.

```
static void processCust(Args _args)
{
    CustTable custTable;
    while select crossCompany Party from custTable
    {
        //..
        //..
        //..
    }

}
```

- Use the pack/unpack design pattern to save or store the state of an object, and then later re-instantiate the same object. The pattern is implemented by using containers. Containers are pass-by-value, so when you pass instances from the client/server tier, you can use packing to avoid holding a proxy object (which will generate an extra RPC each time you call anything on it).

  RunBaseBatch uses the pack/unpack pattern. Be sure to pack/unpack correctly in classes that extend **RunBaseBatch**. For more information about the pattern, see Pack-Unpack Design Pattern.

Example scenario

In this example, pack and unpack are implemented for **RunBaseBatch**.

Bad solution: Incorrect pack and unpack implementation

In this example, pack and unpack are not implemented correctly.

```
class MyCustomEngine extends RunBaseBatch
{

    smmIncludeActivities        includeActivities;
    smmIncludeDocuments         includeDocuments;
    smmIncludeResponsibilities  includeResponsibilities;
    smmIncludeCampaigns         includeCampaigns;
    smmIncludeContacts          includeContacts;
    smmIncludeProjects          includeProjects;


    #define.CurrentVersion(1)
    #localmacro.CurrentList
    #endmacro
}

public container pack()
{
    return connull();
}

public boolean unpack(container packedClass)
{
    return true;
}
```

Good solution: Correct pack and unpack implementation

In this example, pack and unpack are implemented correctly.

```
class MyCustomEngine extends RunBaseBatch
{
    smmIncludeActivities        includeActivities;
    smmIncludeDocuments         includeDocuments;
    smmIncludeResponsibilities  includeResponsibilities;
    smmIncludeCampaigns         includeCampaigns;
    smmIncludeContacts          includeContacts;
    smmIncludeProjects          includeProjects;

    #define.CurrentVersion(1)
    #localmacro.CurrentList
    includeResponsibilities,
    includeCampaigns,
    includeActivities,
    includeDocuments,
    includeContacts,
    includeProjects
    #endmacro
}

public container pack()
{
    return [#CurrentVersion, #CurrentList];
}

public boolean unpack(container packedClass)
{
    int version     = RunBase::getVersion(packedClass);


    switch (version)
    {
        case #CurrentVersion:
            [version,#CurrentList] = packedClass;
            return true;
        default :
            return false;
    }

    return false;
}
```

Code formatting recommendations

The following table lists formatting best practices and provides examples.

Best practice	Example

Place the opening brace at the beginning of the next line.	`if (someExpression)`
	`{`
	`    doSomething();`
	`}`

Best practice	Example

Align the closing brace with the corresponding opening brace.	`if (someExpression)`
	`{`
	`    doSomething();`
	`}`

Place each opening and closing brace on its own line.	`if (someExpression)`
	`{`
	`    doSomething();`
	`}`

Do not omit braces. Braces are not optional, because they increase code readability and maintainability. They should be included even for single-statement blocks.	`if (someExpression)`
	`{`
	`    doSomething();`
	`}`

Omit braces for **switch** statements. These braces can be omitted, because the **case** and **break** statements clearly indicate the beginning and end.	`case 0:`
	`    doSomething();`
	`    break;`

Use a single space in the following cases: • On each side of an assignment operator	**Correct example:** `cust.Name = "Jo";` **Incorrect example:** `cust.Name="Jo";`

• After the comma between parameters	**Correct example:**
	`public void doSomething(int _x, int _y)`
	Incorrect example:
	`public void doSomething(int _x,int _y)`

• Between arguments	**Correct example:** `myAddress(myStr, 0, 1)` **Incorrect example:** `myAddress(myStr,0,1)`
• Before flow control statements	**Correct example:** `while (x == y)` **Incorrect example:** `while(x == y)`
• Before and after binary operators	**Correct example:** `if (x == y)` **Incorrect example:** `if (x==y)`
• After the semicolon between the parts of a **for** statement	**Correct example:** `for (i = 0; i < 10; i++)` **Incorrect example:** `for (i = 0;i < 10;i++)`
Do not use any spaces in the following cases: • After an opening parenthesis or before a closing parenthesis	**Correct example:** `myAddress(myStr, 0, 1)` **Incorrect example:** `myAddress( myStr, 0, 1 )`
• Between a member name and the opening parenthesis	**Correct example:** `myAddress()` **Incorrect example:** `myAddress ()`
• After an opening bracket or before a closing bracket	**Correct example:** `x = dataArray[index];` **Incorrect example:** `x = dataArray[ index ];`
• Before or after unary operators	**Correct example:** `if (!y)` **Incorrect example:** `if (! y)`

Best practice	Example

Use four spaces as the standard indent. In the code editor, the Tab key inserts four spaces. Indent in the following cases:
• The contents of code blocks
	`if (someExpression)`
	`{`
	`    doSomething();`
	`}`

• Case blocks, even though they do not use braces	`switch (someExpression)`
	`{`
	`    case 0:`
	`        doSomething();`
	`        break;`
	`    …`
	`}`

• A wrapped line one indent from the previous line	`lastAccount = this.doSomething(`
	`    cust,`
	`    firstAccount,`
	`    startDate,`
	`    endDate);`

Wrap lines that are too long to fit on a single line.	
Wrap shorter lines to improve clarity.	

Place each wrapped **select** and **while select** statement keyword at the beginning of a new line. The content associated with each keyword should be indented by one indent under the corresponding keyword.	`select firstonly cust`
	`    where someExpression1`
	`        && someExpression2`
	`        && someExpression3;`
	
	`select count(RecId)`
	`    from cust`
	`    where someExpression1`
	`        && someExpression2`
	`        && someExpression3;`
	
	`while select firstonly cust`
	`    order by Name, AccountNum`
	`    where someExpression1`
	`        && someExpression2`
	`        && someExpression3`
	`{`
	`    …`
	`}`

Best practice	Example

Do not use more or less than four spaces to force special alignment.	**Correct example:**
	`lastAccount = this.doSomething(`
	`    cust,`
	`    firstAccount,`
	`    startDate,`
	`    endDate);`
	Incorrect example (the indent is 14 spaces):
	`last = this.do(`
	`              cust,`
	`              firstAccount,`
	`              startDate,`
	`              endDate);`

Put each indented parameter or argument on a separate line.	
Use **switch** statements instead of consecutive **if** statements.	
Do not use parentheses around the value of the cases of a **switch** statement.	
Do not put the closing parenthesis for a method call on a new line.	

Naming

Best practice	Example
Use Camel Casing for member variables, method names, and local variables.	`serverClass;`
Use Pascal Casing for AOT elements.	`AddressCountyRegion;`
Prefix parameter names with an underscore (_).	`myJob(Args _args)`
Do not use Hungarian notation. Do not encode the type of a variable in its name.	**Incorrect example:** `strName`
Avoid prefixing local variables.	**Incorrect example:** `stringName` or `intCount`
Use meaningful and self-documenting names.	

Object names should not be hard-coded. If the objects are renamed or removed in future version, hard-coded names do not result in compilation errors and are very difficult to find.

It is always better to wrap object names in methods such as **formstr(..)** and **tablestr(..)**, because they result in a compilation error and easily surface naming issues.

Example scenario

In this example, object names are required in code.

Bad solution: Hard-coded object names in code

In this example, object names are hard-coded in code.

```
Args = new Args('SalesTable');
methodName = 'runXyz';
```

Good solution: Object names in code

In this example, object names are wrapped with correct methods.

```
Args = new Args(formstr(SalesTable));
methodName = tablemethodstr(SalesTable,runXyz);
```

Code commenting

This section provides best practice guidance for writing code comments. Comments should be used to describe the intent, algorithmic overview, and logical flow. Provide comments so that someone other than the original developer can understand the behavior and purpose of the code. As a best practice, most code has comments reflecting the developer's intent and approach for the code. Use comments liberally. Include comments that indicate who made the changes, when the changes were made, why the changes were added, and what the changes do. Comments are particularly beneficial when multiple parties are involved in modifying and maintaining code.

Best practice	Example
Do not use comments that repeat the code.	

Do not use multi-line syntax, `/* … */`, for comments. Single-line syntax, `// …`, is preferred, even when a comment spans multiple lines.	`public int getCount()`
	`{`
	`    ;`
	
	`    // This comment spans multiple`
	`    // lines because it has`
	`    // a lot to say. The use of`
	`    // multi-line syntax is`
	`    // not allowed.`
	`    …`
	`}`

Do not place comments at the end of a line, unless the comment is very short. In most cases, comments should be placed above the code.	`public class ArrayList`
	`{`
	`    int count; // -1 indicates uninitialized array`
	`}`

Remove TODO comments well in advance of a release.	`// TODO: The TODO comments should start with TODO.`

Example scenario

Often, developers comment code assuming that they will look at it later. Ensure that such comments are identifiable by using the notation TODO rather than normal comments.

Bad solution: To do not called out in a comment

```
// Will handle this later
/*
Select * from…
*/
```

Good solution: To do called out in a comment

```
// TODO  - Need to handle this later.
/*
Select * from…
*/
```

XML documentation

Use XML documentation to provide information about usage, to help other programmers decide whether they want to use a method. The following list provides best practice guidance for XML documentation:

- Add XML documentation with meaningful content.
- Use XML documentation to provide users and potential users with the information they need.
- Do not use XML documentation to discuss implementation details or other items not related to use.
- Do not add XML documentation for the sake of improving code coverage.
- Be aware of the methods that are used for automatically generated XML documentation – for example, **New** and **construct**.

Labels and text

The following list provides best practice guidance for labels and text:

- Use labels for text that will appear in the user interface.
- Enclose labels in double quotation marks.
- Do not concatenate multiple labels.
- Use single quotation marks for text that will not appear in the user interface.
- Microsoft reserves the right to change labels in future releases.
- Make sure that labels communicate a single idea.

Database

The following list provides best practice guidance related to the database:

- Do not use direct SQL calls from X++ code. X++ SQL respects security and other framework features. Direct SQL calls do not respect Microsoft Dynamics AX security and other frameworks.
- Include a **try catch** around all transactions that could result in deadlock. Make sure that the **try** for a deadlock is idempotent. In other words, no matter how many times the **try** is attempted, it yields the same result.
- Consider clarity when deciding on the number of **return** statements in a method.
- Avoid display methods whenever possible.
- Often, update conflicts are handled in code. However, they are not always implemented correctly. For example, when an update conflict occurs, and a retry is performed, you must reset the query and start again.
- Run code on AOS whenever possible.
- Use **where** clauses that align with indexes in **select** statements and queries.
- If method calls are used to test conditions, put the method calls after the other conditions. In this way, if the other conditions fail, you will not incur the cost of running the method.
- Consider specifying a field list in **select** statements to increase performance.
- Use **firstonly** where applicable to increase performance. If you are going to use only the first record, or if only one record can be found, the **firstonly** qualifier optimizes the **select** statement for that circumstance.

- Use aggregates in the selection criteria instead of having the code do the aggregation. If aggregations are issued in the **select** statement instead of in code, the processing is done at the database server, which is much more efficient.
- Use table joins instead of nested **while** loops. Whenever possible, use a join in the **select** statement instead of using a **while** loop and then using an inner **while** loop on the related table. This reduces the amount of communication between AOS and the database.
- In an X++ SQL statement, if a method call is part of the where clause, it is only executed once before the query is executed. It will not be executed for each row.

Example scenario

In this example, a **select** statement on a buffer is done, and in same statement, a method is executed on same buffer.

Bad solution: Method execution in the select statement

In this example, the **select** statement has a method call on the selected buffer.

```
While select * from salesTable where salesTable.Amount == salesTable.AmountCalc()
{
    …
}
```

Good solution: Method execution moved out of the select statement

In this example, the **select** statement does not have a method call on the selected buffer.

```
While select * from salesTable
{
    If (salesTable.Amount == salesTable.AmountCalc())
    {
    }
    …
}
```

- X++ SQL **select** statements should result into a deterministic result set, and they should not use filters that would not provide a deterministic result set.
  - For example, ReqPo does not have a unique index on ItemId.

    ```
    Select * from ReqPo where ReqPo.ItemId == xxxx;
    ```
  - If you believe that, for a particular implementation, some logical assumptions are made that would guarantee a deterministic result set, those assumptions should be enforced in the database and/or in code.
- If back-end code is used to insert, delete, or update records, make sure that you explicitly call the related validate methods, such as **validateWrite**, **validateDelete**, and so on. These methods are not called implicitly by the framework when the operations are executed from back-end X++. However, if a UI is used, back-end frameworks call these methods for you.

Example scenario

In this example, the **delete** method on a buffer is called in code.

Bad solution: Delete called without ValidateDelete

In this example, the **delete** method on the buffer is called without calling **validateDelete**.

```
Select forupdate custTable where …;
custTable.delete();
```

Good solution: Delete called with ValidateDelete

In this example, the **delete** method on the buffer is called before **validateDelete** is called.

```
Select forupdate custTable where …;
If (custTable.validateDelete())
custTable.delete();
```

Transactions

Keep the following best practices in mind when creating transactions:

- Keep database transactions as short as possible.
- Do not include user interaction inside database transactions.
- Use **throw** instead of **ttsAbort**.
- Do not run transaction (ttsBegin/ttsCommit blocks on the client, because you can hold transactions open indefinitely.
- On a server-side TTS block, do not call back to the client to display dialog boxes.
- TTS blocks should be defined so that transactional integrity is always maintained. Often, for a particular business process, you have multiple tasks. Each task maps to multiple database operations. When each task is executed, the code should maintain the integrity of the transaction. For example, if you are working on invoicing, multiple tasks could be posting miscellaneous charges, tax transactions, and invoices. While the invoice process is being executed, if miscellaneous charges are posted correctly, but tax transactions were not posted correctly because of an error, the posting of miscellaneous charges should be rolled back when the miscellaneous charges are rolled back. The transactions blocks for the process should be defined so that all the database operations are committed as part of a single transaction.
- For some business scenarios, it is possible to maintain an intermediary state. If you can, we recommend that you divide the process into sub-processes and commit them independently of the parent process. In such cases, if there is a failure at a later stage, part of the process has been successfully executed and committed. When the process is run the next time, it should start from the state at which it was previously committed, rather than from the beginning.

Example scenario

In this example, an operation is committed in multiple transactions.

Bad solution: Multiple transaction blocks for a single transaction

In this example, a single transaction is committed in multiple transaction blocks.

```
Void doOperationXyz()
{
    Ttsbegin;
    …
    …
    Table1.insert();
    …
    …
    Ttscommit;
    While select table2
    …
    {
        Ttsbegin;
        Table3…
        Table3.update();
        Ttscommit;
    }
}
```

Good solution: A single transaction block

In this example, a single transaction is committed in a single transaction block.

```
Void doOperationXyz()
{
    Ttsbegin;
    …
    …
    Table1.insert();
    …
    …
    While select table2
    …
    {
        Table3…
        Table3.update();
    }
    Ttscommit;
}
```

Exceptions

The following list provides best practice guidance related to exceptions:

- Throw an exception to stop the X++ call stack that is currently being executed.
- Include a localized error message with all thrown exceptions.
- Use the info, warning, and error functions without a thrown exception in cases where the X++ call stack that is being executed should not be stopped.
- Use **throw** with the static helpers on the **Error** class, such as **Error::missingParameter** and **Error::wrongUseOfFunction** for errors targeted at developers.
- Do not throw an exception for invalid assumption cases where **Debug::assert** is more appropriate.

Parameters

Text and numeric values related to business processes should not be hard-coded. Instead of hard-coding those values in code, you should parameterize them by using of macros or database fields.

Example scenario

In this example, business values are used in code.

Bad solution: Business values hard-coded in code

In this example, business values are hard-coded in code.

\Classes\SalesFormLetter_Invoice\createPayment

```
/// <summary>
/// Creates payments and prints payment proposals if needed.
/// </summary>
protected void createPayment()
{
    Set journalSet;
    SetEnumerator se;
    CustInvoiceJour custInvoiceJour;
    CustTrans custTrans;

    journalSet = Set::create(SysOperationHelper::base64Decode(
    formletterOutputContract.parmAllJournals()));
    se = journalSet.getEnumerator();

    while (se.moveNext())
    {
        custInvoiceJour = se.current();
        custTrans = custInvoiceJour.custTrans();

        if (CustPaymModeTable::find(custTrans.PaymMode).PaymMode == "Pay Method1")
        {
            CustVendPaymInvoiceWithJournal::construct(custInvoiceJour).run();
        }
    }
}
```

Good solution: Business values in code

In this example, business values required in code are used from a parameter table.

```
/// <summary>
/// Creates payments and prints payment proposals if needed.
/// </summary>
protected void createPayment()
{
    Set journalSet;
    SetEnumerator se;
    CustInvoiceJour custInvoiceJour;
    CustTrans custTrans;

    journalSet =
Set::create(SysOperationHelper::base64Decode(formletterOutputContract.parmAllJournals()));
    se = journalSet.getEnumerator();

    while (se.moveNext())
    {
        custInvoiceJour = se.current();
        custTrans = custInvoiceJour.custTrans();

        if (CustPaymModeTable::find(custTrans.PaymMode).PaymOnInvoice)
        {
            CustVendPaymInvoiceWithJournal::construct(custInvoiceJour).run();
        }
    }
}
```

Example scenario

In this example business values are used in code.

Bad solution: Business values hard-coded in code

In this example, business values are hard-coded in code.

```
static void findInvoiceLocation(Args _args)
{
    DirPartyLocationRole    partyLocationRole;
    LogisticsLocationRole   locationRole;

    select recid from locationRole
        where locationRole.Name == "Invoice"
        join PartyLocation from partyLocationRole
        where partyLocationRole.LocationRole == locationRole.RecId;


}
```

Good solution: Business values in code

In this example, business values are referenced by using an enum.

```
    static void findInvoiceLocation(Args _args)
{
    DirPartyLocationRole    partyLocationRole;
    LogisticsLocationRole   locationRole;

    select recid from locationRole
        where locationRole.Type == LogisticsLocationRoleType::Invoice
        join partyLocation from partyLocationRole
        where partyLocationRole.LocationRole == locationRole.RecId;


}
```

Validate methods

The following section describes best practices for using **validate** methods.

Do not initialize or modify in validation fields

Validate methods, such as **validateField()** and **validateWrite()**, should only be used to modify existing validations or add new validations. Do not write code in **validate** methods that would initialize or modify the values in fields.

Example scenario

In this example, fields are initialized when a related field is modified.

Bad solution: Modifying fields in validateField

In this example, a field is initialized when a related field is validated.

```
boolean validateField(FieldId p1)
{
    boolean                     ret;

    ret = super(p1);

    if (ret)
    {
        switch (p1)
        {
            …
        …


            case fieldNum(CustTable, CreditMAX) :
                if (this.CreditMAX > 1000)
                {
                    this.ABC = ABC::A;
                }
                else
                {
                    this.ABC = ABC::C;
                }

                if (this.CreditMAX < 0)
                {
                    ret = checkFailed("@SYS69970");
                }
                break;


        …
        …
        }
    }
    return ret;
}
```

Good solution: Modifying fields in modifiedField

In this example, a field is initialized when a related field is modified.

```
public void modifiedField(FieldId p1)
{
        boolean                         ret;

        super(_fieldId);

        switch (_fieldId)
        {
            …
        …


            case fieldNum(CustTable, CreditMAX) :
                if (this.CreditMAX > 1000)
                {
                    this.ABC = ABC::A;
                }
                else
                {
                    this.ABC = ABC::C;
                }

                break;


        …
        …
         }
}
```

Do not validate in insert, update, or delete methods

Validation logic should not be included in **insert**, **update**, or **delete** methods. Validations should always be part of **validate** methods.

Example scenario

In this example, validation logic needs to be added before a record is inserted.

Bad solution: Validation logic in an insert method

In this example, validation logic is added to an **insert** method.

```
//MyCustTable.insert
public void insert()
{
    if (this.Amount < this.Discount)
    {
        throw error("Discount cannot be greater than amount.");
    }
    if (this.Discount < 0)
    {
        throw error("Discount cannot be less than zero.");
    }

    super();
}
```

Good solution: Validation logic in a validate method

In this example, validation logic is added to a **validate** method. In this case, on the EDT for **Discount**, you can also set **AllowNegative=No** to enforce the rule that the discount must always be greater than 0 (zero).

```
private boolean validateDiscount()
{
    boolean ret = true;

    if (this.Amount < this.Discount)
    {
        ret = checkfailed("Discount cannot be greater than amount.");
    }
    if (this.Discount < 0)
    {
        ret = checkfailed("Discount cannot be less than zero.");
    }
```

```
        return ret;
    }

    public boolean validateWrite()
    {
        boolean ret;

        ret = super();

        if (ret)
        {
            ret = ret && this.validateDiscount();
        }

        return ret;
    }

    public boolean validateFieldValue(FieldName _fieldName, int _arrayIndex = 1)
    {
        boolean ret;

        ret = super(_fieldName, _arrayIndex);

        if (ret)
        {
            switch (_fieldName)
            {
                case fieldStr(MyCustomTable, Discount) :
                case fieldStr(MyCustomTable, Amount) :
                    ret = ret && this.validateDiscount();
                    break;
            }
        }

        return ret;
    }
```

Return values

Some framework methods return values. If those methods are customized, it is important to make sure that your customized code properly handles the return value.

Example scenario

In this example, the **validateField** method is customized to add more validations.

Bad solution: Adding validations in validateField

In this example, validations are added in the **validateField** method.

```
Public Boolean validateField(FieldId _fieldId)
{
    Boolean ret;
    Ret = super();
    Ret = this.xyz(); // Customized code
    …
    Return ret;
}
```

Good solution: Adding validations in validateField

In this example, validations are added in the **validateField** method.

```
Public Boolean validateField(FieldId _fieldId)
{
    Boolean ret;
    Ret = super();
    Ret = Ret && this.xyz();
    …
    Return ret;
}

Public Boolean validateField(FieldId _fieldId)
{
    Boolean ret;
    Ret = super();
    If (ret)
    {
        Ret = this.xyz();
    …

    }
}
```

Client user interface guidelines

The following section lists the user interface best practices.

Forms

The following section describes best practices for forms.

Do not replace query objects at run time

Do not replace the query object in a form at run time. Replacing an existing query at run time has many disadvantages – for example:

- The framework code behind the query, which performs operations on the underlying query object, is missed on query objects created at run time.
- The ranges and sort order added by the user are lost when a query object is created at run time.

Example scenario

In this example, an existing query needs to be modified.

Bad solution: Replacing the query object

In this example, the entire query object is replaced.

```
void init()
{
    Query                   query;
    QueryBuildDataSource    qbds;
    CustTable               custTable;
    VendTable               vendTable;
    ;

    super();

    switch(element.args().dataset())
    {
        case tablenum(CustTable) :
                custTable       = element.args().record();
                query           = new Query();
                qbds            = query.addDataSource(tablenum(CustomTable));
                qbds.addRange(fieldnum(CustomTable,  CustAccount)).value(custTable.AccountNum);
        ..
        ..
                this.query(query);
                break;


        case tablenum(VendTable) :
                vendTable       = element.args().record();
                query           = new Query();
                qbds            = query.addDataSource(tablenum(CustomTable));
                qbds.addRange(fieldnum(CustomTable,  VendAccount)).value(vendTable.AccountNum);
        ..
..
                this.query(query);
                break;
    }
}
```

Good solution: Modifying the query object

In this example, the existing query is modified.

```
void init()
{
    Query                   query;
    QueryBuildDataSource    qbds;
    CustTable               custTable;
    VendTable               vendTable;
    ;

    super();

    switch(element.args().dataset())
    {
        case tablenum(CustTable) :
                custTable       = element.args().record();
                query           = this.query();
                qbds            = query.addDataSource(tablenum(CustomTable));
                qbds.addRange(fieldnum(CustomTable,  CustAccount)).value(custTable.AccountNum);
        ..
        ..
                break;


        case tablenum(VendTable) :
                vendTable       = element.args().record();
                query           = this.query();
                qbds            = query.addDataSource(tablenum(CustomTable));
                qbds.addRange(fieldnum(CustomTable,  VendAccount)).value(vendTable.AccountNum);
        ..
    ..
                break;
        }
    }
```

Modifying framework methods

The following section lists best practices for modifying framework methods.

Be careful where code is placed

If you override framework methods, be careful where you place your code. It is important to identify whether you should place code before or after the **super()** call in the method.

Bad solution: Modifying the query object in executeQuery()

In this case, a range value is added after the query is executed, so the results are not reflected until the query is updated or executed again.

```
Void executeQuery()
{
    Super();
    queryBuildDataSourceRange.value(..);

}
```

Good solution: Modifying the query object in executeQuery()

In this case, the range value is added before the query is executed, so the results are reflected when the query is updated or executed again.

```
Void executeQuery()
{
    queryBuildDataSourceRange.value(..);
    Super();

}
```

Do not comment out the super() method

Do not comment out the **super()** call when you add customizations. The **super()** method calls framework code that performs critical activities that it is not always possible to replicate in customized X++ code.

Example scenario

If at run time on a form, the design is modified so that controls are made visible or invisible, it is advised that you wrap the code inside **element.lock()** and **element.unlock()**, so that there is no flicker when changes are made to the design at run time.

Design forms so that they are not expensive to open

The following list provides design guidance for forms:

- Do not include much code in the **init** method. A common bad practice is to insert a lot of data before a form is opened.
- Make the default view a cheap query. Expensive queries should be available, but they should not be loaded by default.
- The number of controls in a form plays a significant role in the initialization performance of that form. Do not add controls that are rarely used. Remove controls from SYS forms if your business does not use them.

Bad solution: Heavy operations in the init method

In this example, heavy code is written in **init()**, which will slow down loading of the form.

```
/// <summary>
/// Initialize the form.
/// </summary>
public void init()
{
    SalesTable  salesTable;
    SalesTotals salesTotals;

    MyCustomTable customTable;
    Amount     amount;
    super();

    while select * from salesTable
    {
        salesTotals = SalesTotals::construct(salesTable);
        salesTotals.calc();
        amount += salesTotals.totalAmount();

        customTable.clear();
        customTable.ABC = salesTotals.totalAmount();
        //..
        //...
        customTable.insert();
    }
}
```

Queries to use in forms

The following section lists best practices for queries in forms.

- Consider using passive joins if you must support an expensive query or data that is not shown in the default view – for example, when data is shown on another tab.
- If data related to multiple tables is shown in a single grid, make sure that a single query is used to fetch that data. This would require the join type to be **Inner/Outer**.

- If ranges are added in code for Query objects, they should be locked or hidden, depending on the need, so that the end user does not change them.

Example scenario

If a particular form is showing records for the current user only, make sure that a range is added for that form.

Bad solution: The range is not locked

In this example, a particular form is showing records for the current user only, but the range for the form is not locked.

```
void init()
{
    QueryBuildDataSource    qbds;

    super();

    qbds = this.query().dataSourceTable(tablenum(ContactPerson));
    qbds.addRange(fieldnum(ContactPerson,
MainResponsibleWorker)).value(queryValue(currentWorker()));

}
```

Good solution: The range is locked

In this example, a particular form is showing records for the current user only, and the range for the form is locked.

```
    void init()
{
    QueryBuildDataSource    qbds;
    QueryBuildRange         qbRange;

    super();

    qbds = this.query().dataSourceTable(tablenum(ContactPerson));
    qbRange = qbds.addRange(fieldnum(ContactPerson, MainResponsibleWorker));
    qbRange.value(queryValue(currentWorker()));
    qbRange.status(RangeStatus::Hidden);

}
```

What not to include in forms

The following list describes the type of logic to exclude from forms:

- Do not include business logic in UI elements such as forms and reports. These elements should only include UI-specific code. Business processes and logic should be written in classes, with client-independent code that can be used for the Windows client or Enterprise Portal.
- Forms are UI elements that are always executed on the client tier. Do not write code that leads to server-side calls on forms. If you must call server-side APIs, bundle the calls together, so that you make the minimum number of cross-tier calls.

Design data source calls so that they are not expensive

The following list describes how to design data source calls so that they are not expensive.

- Set **OnlyFetchActive=Yes** on data sources if no fields from a table are being used in X++. This optimizes the amount of data that is fetched from the database and transmitted over the wire.
- Set **AutoSearch=No** on infrequently used data sources. Only execute the query when the data is displayed.
- Enable or disable controls in a single RPC call to the server.
- Minimize the work that you do in the **active()** method of a data source.

Role-based security

It is important to understand security within Microsoft Dynamics AX, because incorrectly setting security can significantly affect query performance. This section reviews general security information.

General security guidance from Microsoft is available here: Writing Secure Code.

Key concepts

Security roles

Security roles represent a behavior pattern associated with a defined set of application access privileges. A user can be assigned to one or more security roles, based on that user's job position.

Duties

A duty is a responsibility to perform one or more tasks or services. In the security model, a duty is a set of application access privileges that a user requires to carry out his or her responsibilities. Duties are designed with a specific business objective in mind.

Process cycles

Process cycles organize duties and access privileges according to high-level processes.

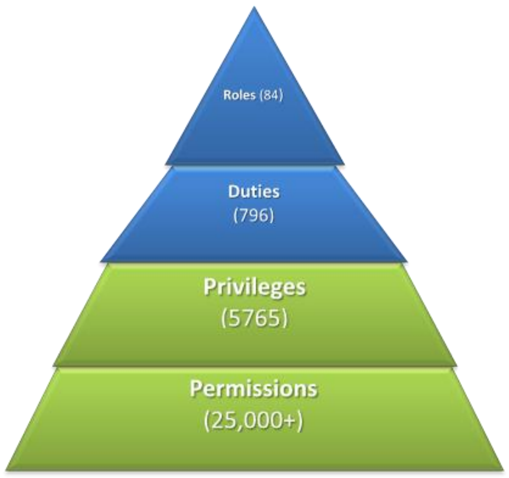


Figure 14 Numbers of roles, duties, privileges, and permissions

Design-level security process

1. Design most functionality so that it can be used by non-administrator roles.
2. Use the principles of segregation of duties and least privilege to determine which security roles have access to functionality.
3. Evaluate the use of table permissions framework (TPF) tables for each role.
4. Develop the application by using recommended coding patterns.
5. Create or update entry points and permissions.
6. Create or update role, duty, and privilege definitions.
7. Link entry points to privileges.

Creating role definitions

1. Enumerate the application use case scenarios to determine the needed roles.
2. Enumerate menu items and other entry points.
3. Fill out a matrix to describe which role needs access to which entry point at what access level.
   - Each row lists the needed privileges.
   - Each column lists the privileges needed for the given role.

Deployment security process

1. Review the definitions of roles and duties.
2. Customize roles and duties as appropriate.
3. Segregate duty rules.
4. Assign users to roles (dynamic or static assignments).
5. Create and maintain overrides at the role level.

Security decisions to make when architecting your solution

Determine whether you should customize out-of-the-box role definitions or create new ones. Use the following steps:

- Evaluate out-of-the-box roles and role definitions.
- Create new roles for scenarios not covered by the base application.
- Create new privileges and/or duties for custom applications.
- Add new privileges and duties in role definitions.

Determine whether you need to fine-tune the out-of-the-box role definitions for your organizations.

- Refactor role definitions into roles and sub-roles.
- Configure user, role, and organization relationships.

Determine which approach to use to control field-level access.

- By default, most privileges and permissions do not specify field-level access.
- Field-level access can be overridden at either the role level or the privilege level.
  - Role-level overrides have narrower scope.
  - Privilege-level overrides have a broader impact.

Determine whether you need to extend Microsoft Dynamics AX functionality to non-Active Directory users.

- Determine the cost of managing accounts for external vendors and customers.
- Determine which ‘identity infrastructure your external vendors and customers use.

Recommendations

- Use sub-roles to share sets of privileges or duties across multiple roles.
- Use the Security Development Tool to get a deeper understanding of role definitions.

Integration considerations

It is important to understand how to best integrate with Microsoft Dynamics AX, because using the wrong service to integrate can significantly affect performance. This section reviews general integration information.

Important: Inserting data into Microsoft Dynamics AX tables from outside of Microsoft Dynamics AX is not supported. System fields such as **RecId** and **RecVersion** cannot be correctly populated from outside the system. Code that inserts data should always use the available integration options, such as calling services.

Integration services

The following illustration shows the services that are available for integration with Microsoft Dynamics AX.

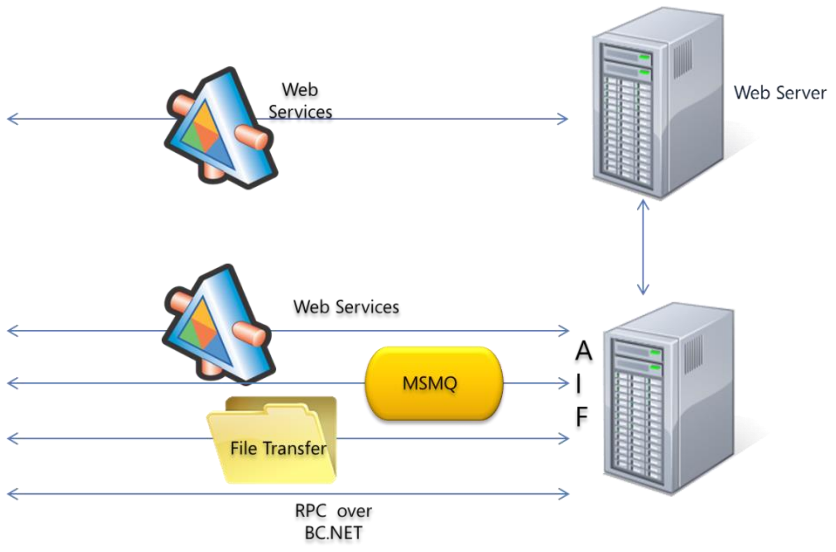


Figure 15 Services that can integrate with Microsoft Dynamics AX

Web services on AOS or IIS

Web services are synchronous, requiring that both applications be running at the same time.

Note: AOS-hosted services only support the Net.TCP protocol.

The characteristics of data transfers include the following:

- They are limited by Windows Communication Foundation (WCF) message size (large).
- They are good when error conditions need to be handled by the client.

For security, Internet-facing services require domain credentials, so some credential mapping is required.

When to use web services:

- Use in transactional scenarios.
- Do not use in cases in which there is fear of denial of service attacks, or fear that the service host may go down.

File transfer

File transfer is asynchronous, so both applications do not need to be running at the same time. Message "batches" can return various error modes.

Security is determined by file ownership.

To ensure security, the sending of files between applications must be done separately from the transfer of data into Microsoft Dynamics AX.

When to use file transfer:

- Use for simple integrations or to test integrations.
- Do not use for cross-geography integrations.
- Use the conversation support feature for messages and batches to help import large amounts of data in priority order.

Message Queuing

Message Queuing (also known as MSMQ) is asynchronous, so both applications do not need to be running at the same time. It has a 4-MB message size limit.

Message Queuing provides access control, audit, and encryption features.

When to use Message Queuing:

- Use for remote write and local read scenarios.
- Use is recommended for publish-subscribe models.
- Use is recommended for scenarios with a very high number of transactions between two applications.

Optimizing and writing code for performance

This section describes best practices for writing performant code.

Working with insert, update, or delete methods for a table

If there is no code in the **insert**, **update**, or **delete** methods for a table, you should remove those methods.

If your code requires initialization for the **insert**, **update**, or **delete** methods from the UI, override the methods in the UI, not at the table level.

Example scenario

In this example, code is required when a record is inserted from a specific form.

Bad solution: Code at back end referencing user interface layer

In this example, code is written in the **insert** method and must be executed when a record is inserted from a specific form.

```
public void insert()
{
    FormDataSource formDs;
    Object         dataSourceObject;


    super();

    formDs = this.dataSource();

    if (formDs && formDs.name() == "MyTableDS1")
    {
        dataSourceObject = formDs;
        dataSourceObject.updateRelatedData();
    }
}
```

Good solution: Customizing at user interface layer

In this example, code is written in the **FormDataSource.write** method and must be executed when a record is inserted from this form. If there is pattern of code like this, you might consider it putting it in a class used for the UI.

```
FormDataSource->write()
public void write()
{
    super();
    this.updateRelatedData();
}
```

Avoid calling client-side code from the server and vice versa

The code being executed on the server side should not call client-side code.

The code being executed on the server side should not have any code related to the UI or user interaction.

Within a transaction block, there should be no code that requires user interaction.

Example scenario

In this example, a client-side API is used.

Bad solution: Client-side API used on server methods

In this example, info log error messages, **startLengthyOperation**, and **endLengthyOperation** are placed in server-side methods such as **insert**, **update**, or **delete**.

```
// Called from client
void run()
{
    MyClass::processOnServer(ParmId);
}

server static public void processOnServer(ParmId _parmId)
```

```
{
    salesParmTable parmTable;
    SalesParmLine  parmLine;
    int            c;

    startLengthyOperation();
    while select * from parmTable
            where parmTable.parmId == _parmId
        join * from parmLine
            where parmLine.ParmId == parmTable.ParmId
    {
        //...
        //...
        // More code related to processing
        //...
        //
        if (parmTable.salesTable().DeliveryDate > (parmTable.Transdate + 30))
        {
            c++;
        }

    }
    endLengthyOperation();
    info(strFmt("@SYS137201",c));
}
```

Good solution: Client-side API separated from server-side code

In this example, client-side code and server-side code are separated to cause a minimum of cross-tier calls. The back-end code being executed on the server side should not have any code related to the UI or user interaction. Similarly, within a transaction block, there should be no code that requires user interaction.

```
// Called from client
void run()
{
    int c;

    startLengthyOperation();

    c = MyClass::processOnServer(ParmId);

    info(strFmt("@SYS137201",c));
    endLengthyOperation();
}

server static public int processOnServer(ParmId _parmId)
{
    salesParmTable parmTable;
    SalesParmLine  parmLine;
    int            c;


    while select * from parmTable
```

```
            where parmTable.parmId == _parmId
        join * from parmLine
            where parmLine.ParmId == parmTable.ParmId
    {
        //...
        //...
        // More code related to processing
        //...
        //
        if (parmTable.salesTable().DeliveryDate > (parmTable.Transdate + 30))
        {
            c++;
        }

    }

    return c;
}
```

Example scenario

In this example, user interaction is used on server-side methods.

Bad solution: Having the Yes/No dialog box (Box::YesNo) in the update method of a table

In this example, a Yes/No dialog box is used in the server-side **update** method.

```
//MyCustomTable.update
public void update()
{
    MyCustomTableLog tableLog;
    TransDateTime    transDateTime = DateTimeUtil::getSystemDateTime();

    ttsBegin;

    super();

    if (Box::yesNo("Insert log?", DialogButton::Yes) == DialogButton::No)
    {
        select forUpdate tableLog where tableLog.MyCustomTable == this.RecId;
        tableLog.MyCustomTable = this.RecId;
        tableLog.TransDateTime = transDateTime;
        tableLog.write();
    }

    ttsCommit;
}
```

Good solution: Remove the Yes/No dialog box (Box::YesNo) in the update method of a table

In this example, a yes/no choice is passed as a parameter.

```
//MyCustomTable.update
public void update(boolean _insertLog = false, transDateTime _transDateTime =
DateTimeUtil::getSystemDateTime())
{
    MyCustomTableLog tableLog;

    ttsBegin;

    super();

    if (_insertLog)
    {
        select forUpdate tableLog where tableLog.MyCustomTable == this.RecId;
        tableLog.MyCustomTable = this.RecId;
        tableLog.TransDateTime = _transDateTime;
        tableLog.write();
    }

    ttsCommit;
}
```

Avoid allowing set-based operations to fall back to row-based operations

Set-based operations such as **update_recordset**, **insert_recordset**, and **delete_from** can fall back to row-based operations at the time of execution because of overridden data methods and database logging. If it is important that you force the use of set-based operations, and then call skip methods, such as **skipData**, before the set-based operation is executed.

Example scenario

In this example, set-based insertion of records is required.

Bad solution: Insert_recordset falling back to a row-based insert

In this example, **insert_recordset** is falling back to a row-based insert, because the **insert** method is overridden.

```
//MyCustTable.insert
public void insert(TransDateTime _transDateTime = DateTimeUtil::getSystemDateTime())
{
    MyCustomTableLog tableLog;

    ttsBegin;

    super();

    tableLog.MyCustomTable = this.RecId;
    tableLog.TransDateTime = _transDateTime;
    tableLog.insert();

    ttsCommit;
}
```

```
void process()
{
    MyCustomTable       customTable;
    InventTable         inventTable;
    MyCustomTableLog    tableLog;
    MyCustomTableLog    existingTableLog;
    TransDateTime       transDateTimeVal = DateTimeUtil::getSystemDateTime();


    ttsBegin;

    insert_recordset customTable(ABC, NameAlias)
        select ABCValue, nameAlias from inventTable;

    ttsCommit;

}
```

Good solution: Insert_recordset is executed with skipDataMethods()

In this example, **insert_recordset** is used with **skipdataMethods**.

```
void process()
{
    MyCustomTable       customTable;
    InventTable         inventTable;
    MyCustomTableLog    tableLog;
    MyCustomTableLog    existingTableLog;
    TransDateTime       transDateTimeVal = DateTimeUtil::getSystemDateTime();

    customTable.skipDatabaseLog(true);
    customTable.skipDataMethods(true);
    customTable.skipAosValidation(true);

    ttsBegin;

    insert_recordset customTable(ABC, NameAlias)
        select ABCValue, nameAlias from inventTable;

    insert_recordset tableLog(MyCustomTable, TransDateTime)
        select RecId, transDateTimeVal from customTable
        notExists join existingTableLog
        where existingTableLog.MyCustomTable == customTable.RecId;

    ttsCommit;

}
```

Design class locations for performance

Business processes modeled in classes should be structured and designed for performance. If basic guidelines are applied, it is easy to structure and write code for performance at design time rather than refactoring code later to make it perform.

Key considerations include the following:

- Use client-server keywords correctly.
- Do not change the default value (**CalledFrom**) of the **RunOn** property of a class without significant analysis. Consider the tier that the object of the class is instantiated in, and the way it is consumed.

Example scenario

If you expect a class to do only database operations, you might change the **RunOn** property on the class to **Server**. However, if you are calling multiple non-static methods of that class from a form on the client tier, each call would result in a cross-tier call.

Bad solution: Marking a class to run on the server

The class is marked to run on the server, which results in cross-tier calls. In this example, for every method we call for myClassObj, there would be one cross-tier call from client to server. This would result in six cross-tier calls.

```
// Method on a Form (running on client side)
Void getInvoiceValue()
{
    AmountMST invoiceValue;
    MyClass myClassObj; // Class marked to Runon Server
    myClassObj = myClassObj::construct(Parm1,Parm2);
    myClassObj.parmVar1(Var1);
    myClassObj.parmVar2(Var2);
    myClassObj.processVars();
    myClassObj.calcResult(parm3,parm4);
    invoiceValue = myClassObj.getInvoiceValue();
}
```

Good solution: Separating client and server code

Separate the code that must run on different tiers. After each tier's code is separated, it can be called by using a static method marked to run on a specific tier. This way, all the related tier code is bundled together, and there is only one cross-tier call for the entire bundle.

```
// Class is marked RunOn - CalledFrom
Class MyProcess extends RunBaseBatch
{
…
…
}

Static void main(..)
{
MyProcess myProcess = MyProcess::construct();
If (myProcess.prompt())
{
myProcess.Run();
}
}
void run()
{
… ..// Any client side or neutral operations. MyProcess::runOnServer(Parm1,Parm2…Parmn); … ..
}
```

```
Server Static void runOnServer(Parm1,Parm2…Parmn)
{
// Db Operations or any server side operations..
}
```

Minimize trips to the database

For **insert**, **update**, or **delete** methods on a table, each call is a trip to the database. Row-based operations in many scenarios can easily be converted to set-based operations. In the case of set-based operations, instead of performing database operation row by row, a single SQL statement is sent to the database for all rows. Use **insert_recordset**, **update_recordset**, and **delete_from** to perform set-based operations.

Example scenario

In this example, multiple records are inserted by using a set-based API.

Bad solution: Row-based insert

In this example, multiple records are inserted by using the row-based **insert** method.

```
 while select salesTable where ….
{
    While select forupdate salesLine…
    {
        salesLine.Field1 = salesTable.Field1.
        …
        ..
        salesLine.update();
    }
}

Update_recordset salesLine setting Field1 = salesTable.Field1 …
    Join salesTable where salesLine…. = salesTable… && ….;
```

Good solution: Set-based insert

In this example, multiple records are inserted by using the set-based method **insert_recordset**.

```
//MyCustTable.insert
public void insert(TransDateTime _transDateTime = DateTimeUtil::getSystemDateTime())
{
    MyCustomTableLog tableLog;

    ttsBegin;

    super();

    tableLog.MyCustomTable = this.RecId;
    tableLog.TransDateTime = _transDateTime;
    tableLog.doInsert();

    ttsCommit;
}

void process()
{
    MyCustomTable       customTable;
    InventTable         inventTable;
    TransDateTime       transDateTimeVal = DateTimeUtil::getSystemDateTime();

    ttsbegin;

    while select ABCValue, nameAlias from inventTable
    {
        customTable.clear();
        customTable.ABC = inventTable.ABCValue;
        customTable.NameAlias = inventTable.NameAlias;
        customTable.insert(transDateTimeVal);
    }

    ttsCommit;
}

void process()
{
    MyCustomTable       customTable;
    InventTable         inventTable;
    MyCustomTableLog    tableLog;
    MyCustomTableLog    existingTableLog;
    TransDateTime       transDateTimeVal = DateTimeUtil::getSystemDateTime();

    customTable.skipDatabaseLog(true);
    customTable.skipDataMethods(true);
    customTable.skipAosValidation(true);

    ttsbegin;

    insert_recordset customTable(ABC, NameAlias)
        select ABCValue, nameAlias from inventTable;

    insert_recordset tableLog(MyCustomTable, TransDateTime)
        select RecId, transDateTimeVal from customTable
        notExists join existingTableLog
        where existingTableLog.MyCustomTable == customTable.RecId;

    ttsCommit;
}
```

Use recordinsert to initialize values

Although set-based operations are preferred, sometimes complex business logic is required to initialize the values before a record is inserted. In those cases, you may not be able to use an **insert_recordset** statement. Instead, use a **RecordInsertList** list, so that the framework can bundle the insert requests together and go to database for the entire bundle rather than for every row.

Example scenario

In this example, multiple records need to be initialized by using business logic before insertion.

Bad solution: Row-based insert

In this example, multiple records are inserted by using a row-based insert.

```
void process()
{
    MyCustomTable       customTable;
    InventTable         inventTable;
    MyCustomEngine      customEngine;
    TransDateTime       transDateTimeVal = DateTimeUtil::getSystemDateTime();


    ttsbegin;
    while select ABCValue, nameAlias from inventTable
    {
        customTable.clear();
        customTable.ABC = inventTable.ABCValue;
        customTable.NameAlias = inventTable.NameAlias;
        customEngine = MyCustomEngine::construct(this.NameAlias, this.ABC);
        if (customEngine)
        {
            this.Amount = customEngine.calc();
        }
        customTable.doInsert();
    }

    ttsCommit;
}
```

Good solution: Insert by using RecordInsertList

In this example, multiple records are inserted by using **RecordInsertList**.

```
void process()
{
    MyCustomTable       customTable;
    InventTable         inventTable;
    MyCustomEngine      customEngine;
    RecordInsertList    recordInsertList = new RecordInsertList(tableNum(MyCustomTable),
true);

    ttsbegin;
    while select ABCValue, nameAlias from inventTable
    {
        customTable.clear();
        customTable.ABC = inventTable.ABCValue;
        customTable.NameAlias = inventTable.NameAlias;
        customEngine = MyCustomEngine::construct(customTable.NameAlias, customTable.ABC);
        if (customEngine)
        {
            customTable.Amount = customEngine.calc();
        }
        recordInsertList.add(customTable);
    }
    recordInsertList.insertDatabase();
```

```
        ttsCommit;
    }
```

Return results in a single SQL query

If you must select data from multiple tables, it is better to use a join option to retrieve your results back in a single SQL query.

Example scenario

In this example, data needs to be fetched from multiple related tables.

Bad solution: Nested selects

In this example, data is fetched by using nested selects.

```
while select salesTable where …
{
    While select salesLine…
    {
        …
        ….
    }
}

While select * from SalesTable where …
{
    Select * from CustTable where …
    Select * from custTrans where …
}
```

Good solution: Select by using a join

In this example, data is fetched by using a join.

```
while select salesTable where ….
    join salesLine where salesLine.XX == salesTable.XX && …
{
}


While select * from SalesTable where …
    Join * from CustTable where …
    Join * from CustTrans where..
{
    ….
}
```

Avoid returning unnecessary results

In SQL statements that involve multiple tables, use an **exists** or **not exists** join whenever possible.

Example scenario

In this example, data is required from a table when a record in a related table exists.

Bad solution: Inner join when a buffer is not needed

In this example, data is required only from salesTable, but because of the inner join with CustTable, CustTable is also fetched.

```
While select * from SalesTable where …
    Join * from CustTable where …
{
    ….
}
```

Good solution: Exists join when a buffer is not needed

In this example, data is required only from salesTable, so an **exists** join is used with CustTable.

```
While select * from SalesTable where …
    exists join from CustTable where …
{
    ….
}
```

Bundle results for cross-tier processing

If you have a requirement to read data from one tier but process it on other tier, it is often confusing how best to design your code for performance. We recommend that you consider bundling results for the best performance.

Example scenario

In this example, you need to read a file line by line from the client side and insert the records for each line into tables.

Bad solution: Mix of client-side and server-side code

If you design a class, insertion happens on the server side for each line, so for each line, you end up doing a server-side call.

```
// File exists on client side and method is called from client side.
public void importFromFile(Filename _name)
{
    #file
    TextIo          textIO;
    MyCustomTable   customTable;
    container       line;
    ABC             abcValue;
    int             numOfRows = 0;

    textIO = new textIO(_name, #io_read);
    if (textIO)
    {
        while (textIO.status() == IO_Status::Ok)
        {
            line = textIO.read();
            if (line)
            {

                customTable.clear();
                customTable.NameAlias = conpeek(line,1);
                abcValue = str2enum(abcValue, conpeek(line,2));
                customTable.ABC = abcValue;
                customTable.doInsert();
                numOfRows++;
            }
        }
    }
    else
    {
        throw error(strfmt("@SYS18678", _name));
    }
    info(strFmt("%1 number of lines imported.", numOfRows));

}
```

Good solution: Communicating cross-tier by using a container

It is better to bundle data by using containers and send it cross-tier rather than sending every line. For this particular scenario, you should read the file line by line and pack it into a container; then, after the container reaches a threshold (for example 1000 lines), you call a server-side static method with this container as a parameter. The server-side method unpacks the container and processes it line by line. In this case, if the file had 10,000 lines, without bundling you would end up with 10,000 server-side calls. With bundling, the operation would result in just 10 calls. However, you need to make sure that you create an appropriate bundle size that is neither very small nor very large.

```
// AOSRunMode::Client
public void importFromFile(Filename _name)
{
    #file
    #define.BundleSize(1000)
    TextIo          textIO;
    container       line;
    container       lineBundle;
    int             numOfRows = 0;

    textIO = new textIO(_name, #io_read);
    if (textIO)
    {
        while (textIO.status() == IO_Status::Ok)
        {
            line = textIO.read();
            if (line)
            {
                lineBundle += [conpeek(line,1), conpeek(line,2)];
                numOfRows++;
                if (numOfRows mod #BundleSize == 0)
                {
                    MyCustomImport::processBundle(lineBundle);
                    lineBundle = conNull();
                }
            }
        }
        if (lineBundle)
        {
            MyCustomImport::processBundle(lineBundle);
            lineBundle = conNull();
        }
    }
    else
    {
        throw error(strfmt("@SYS18678", _name));
    }


    info(strFmt("%1 number of lines imported.", numOfRows));
}

// AOSRunMode::Server
private static server void processBundle(container _lineBundle)
{
    container           line;
    MyCustomTable       customTable;
    ABC                 abcValue;
    int                 i = 0;
    RecordInsertList    recordInsertList = new RecordInsertList(tableNum(MyCustomTable));

    ttsBegin;
    if (i < conLen(_lineBundle))
    {
        line = conPeek(_lineBundle, i);
        customTable.clear();
        customTable.NameAlias = conpeek(line,1);
        abcValue = str2enum(abcValue, conpeek(line,2));
        customTable.ABC = abcValue;

        recordInsertList.add(customTable);
        i++;
    }
    if (i > 0 )
    {
        recordInsertList.insertDatabase();
    }
    ttsCommit;
}
```

Use queries instead of display methods for forms whenever possible

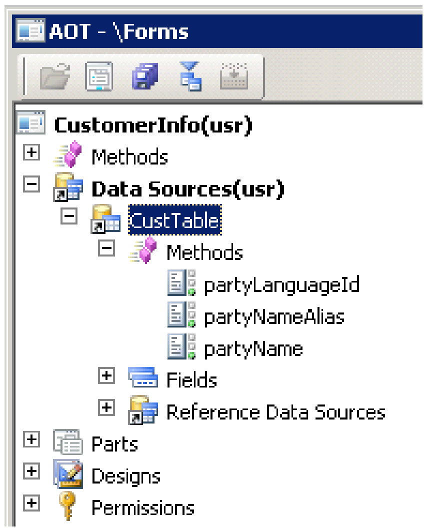
In forms, a common pattern is the need to show some information from a table that is not added as a data source on that form. If multiple fields from a table need to be shown, a developer often ends up creating multiple display methods that perform the same SQL operation and return different field values. In those scenarios, you should evaluate whether it is possible to add the table to the form query and have that record fetched only once as part of same query.

Example scenario

In this example, party data needs to be displayed in a customer information form.

Bad solution: Display methods

In this example, multiple display methods are used to show party information.



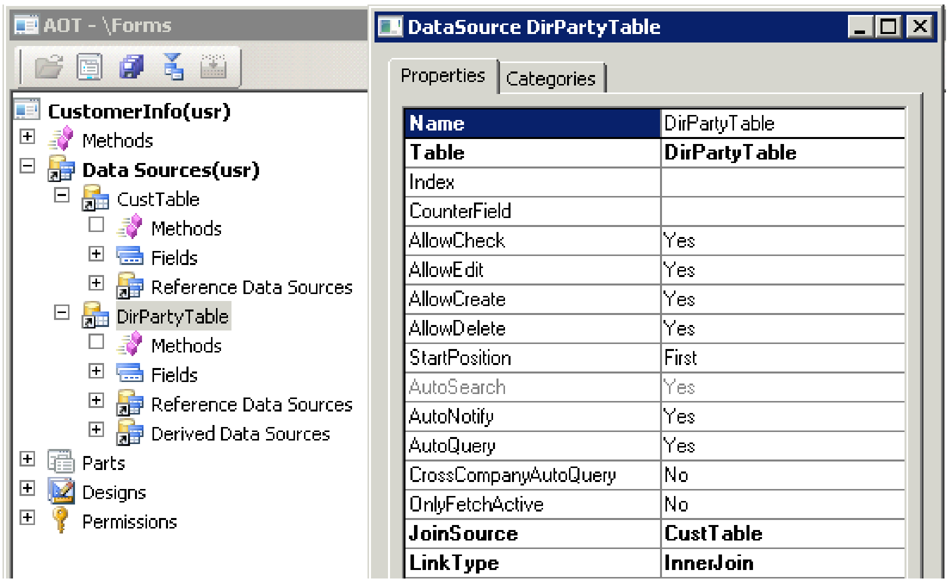
```
CustomerInfo(usr)
  Methods
  Data Sources(usr)
    CustTable
      Methods
        partyLanguageId
        partyNameAlias
        partyName
      Fields
      Reference Data Sources
  Parts
  Designs
  Permissions
```

Figure 16: Multiple display methods for party information

```
display DirPartyName partyName()
{
    return DirPartyTable::findRec(CustTable.Party).Name;
}
display NameAlias partyNameAlias()
{
    return DirPartyTable::findRec(CustTable.Party).NameAlias;
}
display LanguageId partyLanguageId()
{
    return DirPartyTable::findRec(CustTable.Party).LanguageId;
}
```

Good solution: Replacing display methods with joins

In this example, display methods are replaced with a join.



Name	DirPartyTable
Table	DirPartyTable
Index	
CounterField	
AllowCheck	Yes
AllowEdit	Yes
AllowCreate	Yes
AllowDelete	Yes
StartPosition	First
AutoSearch	Yes
AutoNotify	Yes
AutoQuery	Yes
CrossCompanyAutoQuery	No
OnlyFetchActive	No
JoinSource	CustTable
LinkType	InnerJoin

Figure 17: Using a join to display party information

Cache display or edit methods

If display or edit methods are used on grid controls, they can lead to performance issues. It is always advisable to cache display and edit methods.

Specify a field list

While you are writing X++ SQL statements or using query objects, it is important to specify a field list to ensure that not all fields are fetched. Returning the smallest number of fields increases the performance of your code.

It is also important to make sure that, if the table buffer is passed outside of the code block, the consumer of the buffer is aware of which fields were actually fetched. Often, when we do some refactoring or code changes, we tend to assume that all fields were fetched.

Example scenario

In this example, some fields need to be fetched for records.

Bad solution: Select without a field list

In this example, data is fetched without specifying a field list.

```
private boolean validateSalesOrder(SalesId _salesId)
{
    SalesTable              salesTable;
    SalesLine               salesLine;
    SalesLineType_Sales     salesLineType;
    boolean                 ret = true;

    while select salesTable
        where salesTable.SalesId == _salesId
        join salesLine
        where salesLine.SalesId == salesTable.SalesId
    {
        if (salesTable.CashDiscPercent < 0)
        {
            ret = ret && checkFailed("<Error message>");
        }
        if (salesTable.SalesDiscAmount < 0)
        {
            ret = ret && checkFailed("<Error message>");
        }
        if (!salesTable.DeliveryDate)
        {
            ret = ret && checkFailed("<Error message>");
        }
        salesLineType = SalesLineType_Sales::construct(salesLine);
        //
        //salesLineType...
        //..
        //
    }

    return ret;
}
```

Good solution: Select with field list

In this example, data is fetched by using a field list.

```
private boolean validateSalesOrder(SalesId _salesId)
{
    SalesTable              salesTable;
    SalesLine               salesLine;
    SalesLineType_Sales     salesLineType;
    boolean                 ret = true;

    while select CashDiscPercent, SalesDiscAmount, DeliveryDate, SalesId from  salesTable
        where salesTable.SalesId == _salesId
        join salesLine
        where salesLine.SalesId == salesTable.SalesId

    {
        if (salesTable.CashDiscPercent < 0)
        {
            ret = ret && checkFailed("<Error message>");
        }
        if (salesTable.SalesDiscAmount < 0)
        {
            ret = ret && checkFailed("<Error message>");
        }
        if (!salesTable.DeliveryDate)
        {
            ret = ret && checkFailed("<Error message>");
        }
```

```
            salesLineType = SalesLineType_Sales::construct(salesLine);
            //
            //salesLineType...
            //..
            //
        }

        return ret;
    }
```

Processes that are time consuming should have the following characteristics:

- It should be batch-enabled.
- Parallel processing should be considered for isolated portions.
- The Business Operation framework should be considered, because it would provide both service and batch benefits.

In case of complex query patterns that are repeated multiple times, consider pushing the result set to a TempDB temporary table, and use that temp table to join at other places.

Appendix A: Code review checklist

Ensure that code is put through a code review. We recommend that you use the attached checklist.

General information

Projects:	
Version:	
Application build number:	
Models:	
Layers:	
Developers:	
Reviewer:	
Review start date:	
Review end date:	

Results

Description	**Comments**
Code good to check in:	
Reason for acceptance/rejection:	
Design changes required:	
Number of quality issues found:	
Number of performance issues found:	
Major areas of concern:	

Risk and open issues

Functional area	**Description**	**Comments**

Standards

No.	Check	Compliant?	Comments
1	Development is performed on proper layer.		
2	Development is performed on proper model.		
3	Version control is used.		
4	Source control should be configured with the following parameters: **Compiler warnings:** Reject **Compiler errors:** Reject **Compiler to-dos:** Accept (*Accept during development but Reject for final build to production*) **Best practice errors:** Reject **Run Title Case Update:** True		
5	Clean up your code: delete unused variables, methods, and classes.		
6	No objects should be left with their automatically generated names (Class1, Method1, TabPage, Group1, ReportDesign1, Field_1, Field1, and so on).		
7	Standard Microsoft Dynamics AX Best Practices should be followed		
8	The developer guidance document is followed.		

Naming conventions

No.	Check	Compliant?	Comments
1	New objects names are prefixed per guidelines in the developer guidance document.		
2	Meaningful names are used for classes, methods, fields, and so on.		
3	Object names are spelled correctly.		
4	Each path in the AOT is unique; nodes must not have identical names.		
5	All text that appears in the user interface and code must be defined by using a label. It must not be hard-coded.		
6	Do not begin a name with aaa or CopyOf. Do not begin a name with DEL_ unless it is a table, extended data type, or enum, and it is needed for data upgrade purposes.		
7	Use proper casing as defined in the developer guidance document. **Example:** `startBatchJob`		
8	Use Pascal Case naming for AOT elements. **Example:** `CustInvoiceJour`		
9	Prefix parameter names in methods with an underscore (_). **Example:** `findCustByName(Name _name)`		

No.	Check	Compliant?	Comments

10	Use uppercase letters for acronyms that are two characters long or less, and Pascal Case for acronyms containing three or more characters.
	Examples:
	`CustAccountID`
	`VendAccountNum`
	Note: Use the function **Add-Ins** > **Source Code Titlecase Update** to "wash" your code so that it uses the correct case. If you have the **RunTitlecaseUpdate** option turned on in version control, this will happen automatically when you check the object in.		

X++ coding best practices

No.	Check	Compliant?	Comments
11	Methods should be small, concise, and clear. The method should do a single well-defined task.		

12	Placement of braces is correct.
	Incorrect formatting:
	`void myMethod {`
	`    // code`
	`}`
	Correct formatting:
	`void myMethod`
	`{`
	`    // code`
	`}`
	Note: Braces should be placed around every block of statements, even if there is only one statement in the block.		

No.	Check	Compliant?	Comments

13	Indents should be four characters.
	Braces (opening and closing) should be aligned for a code block.
	For **switch-case** statements: indent **case** and **default** statements by one level (with any code within these indented a further level), and indent **break** statements by two levels.
	Example:
	`switch (myEnum)`
	`{`
	`    case ABC::A:`
	`        ...`
	`        break;`
	`    case ABC::B`
	`        ...`
	`        break;`
	`    default:`
	`        ...`
	`        break;`
	`}`
	Indent **where** and other qualifiers to the **select** statement by one level.
	Example:
	`select myTable`
	`    index hint myIndex`
	`    where myTable.field1 == 1`
	`        && myTable.field2 == 2;`		

14	Do not assign values to, or manipulate, actual parameters that are "supplied" by values. You should always be able to trust that the value of such a parameter is the one initially supplied.		
15	There should not be any dead or unreachable code sections.		
16	There is only one statement per line.		
17	Break up complex expressions that are more than one line to make them visually clear.		
18	Do not use parentheses around the case constants.		
19	Add one space between **if**, **switch**, **for**, and **while**, and the expressions starting with parentheses. **Example:** `if (creditNote)`		
20	Use methods on tables before creating new inline SQL.		

Commenting

No.	Check	Compliant?	Comments
21	The XML documentation feature should be used to comment source code.		
22	All comments and expressions should be broken into multiple lines to enhance readability.		

No.	Check	Compliant?	Comments
23	Complex blocks of code should be properly commented.		
24	Avoid using clutter comments, such as an entire line of asterisks. Use white space to separate comments from code.		
25	There should not be any comments that are not relevant to the code.		
26	Comments should not be used to serve as inline translations of the code; instead, they should focus on the intent and function of the code.		
27	All comments should start with an uppercase letter.		
28	Comments should be used to classify the modification type – for example, commenting system changes, adding new lines, and so on.		
29	Reasons should be given if any base layer code is commented.		

Error handling

No.	Check	Compliant?	Comments
30	Exception handling is done.		
31	Never use **infolog.add** directly. Use the indirection methods: **error**, **warning**, **info**, and **checkFailed**.		
32	Design your customizations to avoid deadlocks.		
33	Update conflicts are handled.		

Tables and maps

No.	Category	Check	Compliant?	Comments
34	Metadata	The table must have a label defined from a label file.		
35	Metadata	The table must have a Help text defined from a label file.		
36	Metadata	Title fields should be defined.		
37	Metadata	The table's group property should be set properly, according to its purpose.		
38	Metadata	The table should be attached to a configuration (unless it is always needed). **Note:** New keys may need to be created for the client, based on new functionality added to the system and the level of control the client wants over this functionality.		
39	Metadata	The **CacheLookup** field should be filled in. **Performance note:** This can help increase performance in Microsoft Dynamics AX.		
40	Metadata	The table's **FormRef** property should be set if applicable. **Note:** This is critical for the Go to main functionality.		

No.	Category	Check	Compliant?	Comments
41	Metadata	The table contents should be set if this table contains base or default data.		
42	Metadata Indexes	The table should have a primary key (unique index) defined.		
43	Metadata Indexes	Logical field indexes should be created; these should be based on a common run or large queries from source code. **Performance note:** Indexes are critical to increasing performance in Microsoft Dynamics AX.		
44	Metadata Indexes	The **Cluster Index** field should be filled in if there is only one index on the table, or if there is one index that is used to access this table 90 percent or more of the time. **Note:** Cluster indexes have some overhead when they write data but can be very fast when reading data. **Performance note:** Cluster indexes can help improve performance in Microsoft Dynamics AX.		
45	Field Metadata	All fields that are part of the primary key or index should be mandatory and should not be editable.		
46	Field Metadata	All fields should extend from an extended data type and should be part of at least one field group.		
47	Field Metadata	The field must have a label defined from a label file.		
48	Field Metadata	The field must have a Help text defined from a label file.		
49	Field Groups	The **Auto Report** field group must contain at least one field.		
50	Field Groups	All new field groups must have a label defined from a label file.		
51	Field Groups	Additional field groups should be created to logically group similar fields that can be used in forms and on reports. **Note:** When future additions are made to the table, this will make it easier for you to update forms and reports without having to change those objects.		
52	Delete Actions	To ensure referential integrity between tables (that is, orphaned records are not left in a details table when the header is deleted), you must set the delete action on the table. **Note:** The different delete actions act differently, so based on the desired functionality, you should set this accordingly.		
53	Methods	Static **find** and **exist** methods should exist for the table's primary search key.		
54	Methods	A static **find** method should exist that fetches other related tables by using the foreign key contained in the table.		

Views

No.	Category	Check	Compliant?	Comments
55	Properties	The view must have a label defined from a label file.		
56	Properties	Title fields should be defined.		
57	Properties	The view’s group property should be set properly, according to its purpose.		
58	Properties	The view should be attached to a configuration key and a security key. **Note:** New keys may need to be created for the client, based on new functionality added to the system and the level of control the client wants over this functionality.		
59	Properties	The view’s **FormRef** property should be set if applicable. **Note:** This is critical for the Go to main functionality.		
60	Field Properties	The field must have a label defined from a label file.		
61	Field Properties	The field must have a Help text defined from a label file.		
62	Field Groups	The **Auto Report** field group must contain at least one field.		
63	Field Groups	All new field groups must have label defined from a label file.		
64	Field Groups	Additional field groups should be created to logically group similar fields that can be used in forms and on reports. **Note:** When future additions are made to the view, this will make it easier for you to update forms and reports without having to change those objects.		

Extended data types

No.	Category	Check	Compliant?	Comments
65	Properties	The EDT must have a label defined from a label file.		
66	Properties	The EDT must have a Help text defined from a label file.		
67	Properties	The configuration key should be defined.		

Base enums

No.	Category	Check	Compliant?	Comments
68	Properties	The base enum must have a label defined from a label file.		
69	Properties	The base enum must have a Help text defined from a label file.		
70	Properties	The configuration key should be defined.		
71	Properties	All elements must have a label defined from a label file.		

Configuration and security keys

No.	Category	Check	Compliant?	Comments
72	Properties	The configuration and security keys must have a label defined from a label file.		
73	Properties	Security keys for new modules should be created in a similar fashion and construction to standard Microsoft Dynamics AX modules, based on a menu structure and sections (for example, **Daily**, **Inquiries**, and **Reports**).		

Classes

No.	Category	Check	Compliant?	Comments
74	Methods	All text referenced in the class should be defined from a label in a label file, (for example, messages and Infolog information). Text constants should not be used.		
75	Methods	Check for good SQL syntax. **Performance note:** SQL syntax and construction are primary drivers for performance in Microsoft Dynamics AX.		
76	Methods	Ensure that you are using the proper class type construct.		
77	Methods	Check for and remove unnecessary comments.		
78	Methods	Remove unused variables that are part of the modification layer.		
79	Properties	The **RunOn** property should be set appropriately, based on the classe's functionality.		

Table collections

No.	Category	Check	Compliant?	Comments
80	General	Use table collections if appropriate. If they are used, all FKs inside a table collection should be resolved in the table collection.	Choose an item.	NA

Macros

No.	Category	Check	Compliant?	Comments
81	General	You must only use macros to declare constants or libraries of constants that should be available everywhere in the application.	Choose an item.	NA

Forms

No.	Category	Check	Compliant?	Comments
82	Methods	Avoid putting any code in the **Active** method.		
83	Data Sources	The name of the data source should be the same as the table name. **Note:** When the same table is used multiple times in a form, the name should reflect the purpose of that table in the form.		
84	Design	A caption must be defined from a label file.		
85	Design	The form should have a title data source defined.		

Reports

No checks are currently identified.

Queries

No checks are currently identified.

Jobs

No.	Category	Check	Compliant?	Comments
86	General	Jobs should not be used in production code.		

Menus and menu items

No.	Category	Check	Compliant?	Comments
87	General	All runnable elements you create must have a corresponding menu item with the same name as the element.		
88	Properties	A label must be defined from a label file.		
89	Properties	Help text must be defined from a label file.		
90	Properties	Ensure proper use of the **RunOn** property.		

Microsoft Dynamics is a line of integrated, adaptable business management solutions that enables you and your people to make business decisions with greater confidence. Microsoft Dynamics works like and with familiar Microsoft software, automating and streamlining financial, customer relationship and supply chain processes in a way that helps you drive business success.

U.S. and Canada Toll Free 1-888-477-7989
Worldwide +1-701-281-6500
www.microsoft.com/dynamics

Microsoft