MODULE 4: POSTING

Module Overview

Transactional systems, such as Microsoft Dynamics NAV 2013, record past business events or transactions, and must safeguard the integrity of that information. Some examples of these business events are as follows:

- Purchases from vendors
- Sales to customers
- Consumption of raw materials in production
- Output of finished goods in production
- Usage of resources
- Payments from bank accounts to vendors

To make sure that information about past business events is always intact, Microsoft Dynamics NAV 2013 distinguishes between working data and posted data. *Working data* represents information about current or future transactions. Users can insert, change, or delete that information as needed. *Posted data* represents information about past business transactions. Users cannot insert, change, or delete that information. *Posting* is a process that moves the data from working tables into posted tables.

All functional areas of Microsoft Dynamics NAV 2013 provide very similar features for enabling users to enter the transaction data and process it. This similarity exists at all levels: user interface, data model, and process level. When you develop a new functional area, you must follow the standard concepts as much as possible to maintain a consistent user experience across the application.

Working tables in Microsoft Dynamics NAV 2013 consist of the following:

- Document tables
- Journal tables

Posted tables in Microsoft Dynamics NAV 2013 consist of the following:

- Posted document tables
- Ledger entry tables
- Register tables

There are two posting routines that move the data between these tables:

- Document posting routine
- Journal posting routine

Each of these routine comprises several codeunits.

The Seminar module now contains master tables and document tables to create registrations. The next step is to use the registration information to create ledger entries for seminars through posting routines.

Objectives

The objectives are:

- Explain the working and posting tables.
- Explain posting routines and their relationships.
- Create journal posting routines.
- Create document posting routines.
- Present the best practices for documenting changes to existing objects.
- Program for low impact on the application.

Prerequisite Knowledge

Before you begin to work on posting, it is important to know about journal tables, ledger tables, and some of the elements that are involved in posting.

Journal, Ledger and Register Tables and Pages

Journal tables, ledger tables, and posting codeunits are at the core of every posting process in Microsoft Dynamics NAV 2013.

Journal Tables

A journal is a temporary work area for the user. Users can insert, change, and delete all records in journals. A journal consists of three tables.

Table	Remarks
Journal Template	Journal templates represent transaction types, such as sales, cash receipt, inventory, or reclassification. There is typically only one journal template per transaction type, but users may decide to define more.
Journal Batch	Batches may represent various logical subtypes of the same transaction type. For example, users may have different cash receipt batches for different bank accounts or customer groups, or different inventory batches for different locations or item types. Sometimes, batches represent different users who use them to physically separate transactions that are entered by different users.
Journal Line	Journal line tables store the information about the transaction itself.

Lines belong to batches, and batches belong to templates. The “Journal Structure” figure shows how journal tables are related to one another.

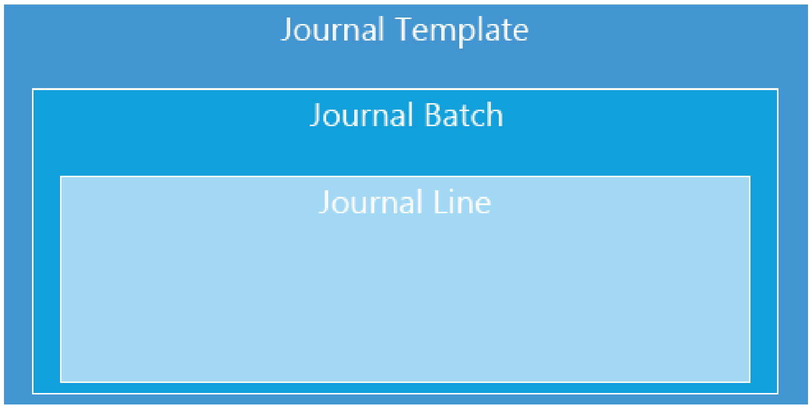


FIGURE 4.1: JOURNAL STRUCTURE

The Journal Page

The primary page to enter information into journals is called by the transaction type, and followed by the word *Journal*, for example: **Sales Journal**, **Cash Receipt Journal**, **Resource Journal**, or **Consumption Journal**. The page is of type worksheet, and uses **Journal Line** as its source table.

The primary key of a **Journal Line** table is a composite key, and consists of the **Journal Template Name**, **Journal Batch Name** and **Line No.** fields. The user never enters information into any of these fields directly. Instead, the **Journal** page sets the field according to the following rules:

- The **Journal** page that the user accesses sets the **Journal Template Name** field. If there are more templates for the same **Journal** page, then users must select the template when they start the **Journal** page. They cannot change the template unless they close and then reopen the **Journal** page.
- The journal page also sets the **Journal Batch Name** field. However, the user may change the **Batch Name** field at the top of the page.
- The **Line No**. field keeps each record in the same template and batch unique. The batch page sets the **Line No.** field automatically through the AutoSplitKey property.

The **Journal** page lets users enter and edit the journal lines that will later be posted into ledger tables. As long as the lines are in the journal, users can freely change or delete them, and they have no effect until the user posts the journal. Users can even leave the lines in the journal table indefinitely.

Ledger Tables

At the core of most functional areas in Microsoft Dynamics NAV 2013, there is a ledger table that keeps transaction history for that functional area. The ledger table is always called *Ledger Entry*. The **Ledger Entry** table is noneditable. Records in it are permanent and users cannot delete or change them, except in specific situations and by using special objects. You also cannot insert the entries directly into a ledger table. You can insert new entries into a ledger table only through a posting routine that moves the data from journal tables to ledger tables. After you post, the lines that were posted are deleted from the journal tables.

The primary key of every ledger table is the **Entry No.** field. There are many secondary keys, and most are compound. These keys are used by reports, pages, and FlowFields.

For most functional areas, there is a link between the **Ledger Entry** tables and the **General Ledger Entry** table. Because of this link, any modifications that you make directly to a ledger table can cause serious problems. Usually, the only way to undo such changes is to restore the most recent backup of the database.

The Ledger Entries Page

A page that shows the records from the **Ledger Entry** table is a List page, and is named after the ledger, followed by the words **Ledger Entries**, for example, **General Ledger Entries**, **Customer Ledger Entries** or **Item Ledger Entries**.

The **Ledger Entries** pages are typically noneditable, and do not allow insertions, modifications, or deletions. However, depending on the transaction type, they may allow certain changes that are typically related to business process specifics. For example, the **Customer Ledger Entries** and **Vendor Ledger Entries** pages allow changes to certain fields to provide putting entries on hold, or to manage the payment discounts after posting.

Note: *You do not protect the* ***Ledger Entry*** *tables directly by making the table fields noneditable. Instead, you must make sure that every page protects the table against unauthorized changes according to the business process requirements for the ledgers.*

The Register Table and Page

Each functional area that includes a ledger also includes a register. A *register* is a table that keeps the history of all transactions. It is the core of the audit trail for the functional area. The table is always named after the ledger, followed by the word *Register*, for example **G/L Register**, **Item Register**, or **Resource Register**. The primary key is always the field **No.**

The **Register** table keeps the summary for the transaction, whereas the **Ledger Entry** table keeps the details for the transaction. There may be multiple ledger entries for each register line. For each transaction, the **Register** table always keeps track of the first and the last Ledger Entry record that is posted by the transaction. The **Register** table also keeps track of the **Creation Date**, **Source Code**, **User ID** and **Journal Batch Name** for the transaction.

For each **Register** table, there is a page that shows the records from the table. This is always named after the ledger, followed by the word *Registers*. The **Registers** page is a noneditable list page for the **Register** table, and always has the same name as its source table.

Every **Registers** page provides the quick means to show the ledger entries that result from the selected transaction in the register. The action is called after the ledger or the sub-ledger that it shows. For example, in the **Item Registers** page, there are **Item Ledger**, **Phys. Inventory Ledger**, **Value Entries**, and **Capacity Ledger** actions. Each of these actions runs a separate codeunit that receives the **Register** record, filters the ledger entries according to the **From Entry No.** and **To Entry No.** fields, and then shows the appropriate **Ledger Entries** page. This codeunit is always called after both the register page, and the ledger it shows. For example, clicking the **Item Ledger** action calls the Item Reg.-Show Ledger codeunit.

Journal Posting Codeunits

For each journal type, there is a group of codeunits that is responsible for moving the data from the journal tables into the ledger tables. These codeunits also make sure that all the data that is moved into the ledger is correct for each line and for the entire table. That group of codeunits is frequently called a *posting routine*. A posting routine performs the following tasks:

- Takes journal lines and checks them.
- Converts journal lines to ledger entries.
- Inserts journal lines into the ledger table.
- Makes sure that all posted transactions are consistent.

Although there are many types of posting routines in Microsoft Dynamics NAV 2013, they all follow the same data structure and architectural principles.

The Post Line Codeunit

The primary codeunit that does the work of posting for a particular journal is named after the journal name followed by the words *Post Line*, for example Gen. Jnl.-Post Line or Res. Jnl.-Post Line. The primary goal of a Post Line codeunit is to transfer the information from the **Journal Line** table into the **Ledger Entry** table, although it also performs other functions, such as calculations and data checking.

Note: *Depending on the business process that it handles, the Post Line codeunit may even post to multiple ledgers at the same time. For example, the Gen. Jnl.-Post Line codeunit posts information into general, customer, vendor, bank account, and fixed asset ledgers.*

Journal Posting Companion Codeunits

For each type of posting routine (General Ledger, Item, Resource, and so on), the Post Line codeunit has two companion codeunits.

Codeunit	Purpose
Check Line	Checks each journal line before it is posted. It receives the journal line as a parameter, and never reads it from the database. Check Line may read the related data from the database, however, it never writes any data back to the database. It checks for any conditions that may cause the posting to fail. It runs before the posting process starts to make sure that the posting process does not begin if there are any errors. The posting process in the Post Line codeunit performs many write operations. It also adds many locks, some of them explicit, so that the Check Line guarantees the highest possible concurrency between transactions. The posting process causes the problematic journal to fail before any locks are added. This codeunit is called by the Post Batch codeunit, but also by the Post Line codeunit.
Post Batch	The Post Batch codeunit repeatedly calls the Check Line codeunit to check all lines. If this check succeeds, then Post Batch repeatedly calls the Post Line codeunit to post all lines. The Post Batch codeunit is the only one that actually reads or updates the **Journal** table. The other codeunits use the **Journal** record that is passed into them. In this manner, you can call the Post Line codeunit directly from another posting codeunit without having to update the **Journal** table. The Post Batch codeunit is called only when the user clicks **Post** within the **Journal** page.

By convention, the last digits of the object ID numbers of the posting codeunits are standardized.

Posting Codeunit	Ends with	Example
Check Line	1	11 Gen. Jnl.-Check Line
Post Line	2	12 Gen. Jnl.-Post Line
Post Batch	3	13 Gen. Jnl.-Post Batch

These codeunits do not require any user input. This is because they can be called from other objects that are part of larger batch processes, or from outside Microsoft Dynamics NAV using web services. In these situations, the user interface is either not desirable or not possible. The Post Batch codeunit displays a dialog that shows the posting progress and lets the user cancel the posting.

Journal Posting Starter Codeunits

Posting is a complex, and frequently time-consuming process that requires exclusive access to the data. Therefore, it must run without interruption so that posting codeunits do not allow any kind of user interaction. If there is any input that must be provided to the posting process, users must provide that input at the very beginning of the process.

Any user interaction during posting is handled by another set of codeunits:

Codeunit	Description	Object ID ends with	Example
Post	Asks the user whether to post, and then calls Post Batch.	1	231, Gen. Jnl.-Post
Post + Print	Asks the user whether to post, then calls Post Batch, and then calls the Register Report.	2	232 Gen. Jnl.-Post+Print
Batch Post	Asks whether to post the selected batches and then repeatedly calls Post Batch for each selected batch. Users can run this codeunit only from the **Journal Batches** page.	3	233, Gen. Jnl.-B.Post

Codeunit	Description	Object ID ends with	Example
Batch Post + Print	Confirms that the user wants to post the selected batches, then calls Post Batch for each selected batch, and then calls the Register Report.	4	234, Gen. Jnl.-B. Post+Print

The Journal Posting Process

The journal posting process involves one of the following starter codeunits:

- Post Batch
- Check Line
- Post Line

These three codeunits are the most important components of any posting routine, because they run the bulk of the business logic of transaction posting for a functional area.

Check Line Codeunit

As its name suggests, the Check Line codeunit checks the Journal Line that is passed to it. It does so without reading from the database server.

Before checking any of the fields, this codeunit makes sure that the journal line is not empty. It does so by calling the **EmptyLine** function in the **Journal** table. If the line is empty, the codeunit skips it by calling the **EXIT** function.

The last thing that the codeunit verifies is the validity of the dimensions for the journal line. The codeunit does so by calling the DimensionManagement codeunit. If the codeunit does not stop the process with an error, then the journal line is accepted, and the posting continues.

Post Line Codeunit

The Post Line codeunit is responsible for actually writing the journal line to the ledger. It only posts one journal line at a time, and it does not examine previous or upcoming records.

The function that runs the bulk of work in this codeunit is the **Code** function.

The OnRun trigger of the Post Line codeunit is usually never called, but it was called in earlier versions of the product, and is retained for backward compatibility. Instead, other codeunits call the **RunWithCheck** function that first calls the Check Line codeunit, and then calls the **Code** function.

Like the Check Line codeunit, this codeunit skips empty lines by exiting. This guarantees that empty lines are not inserted into the ledger. The first thing the codeunit does if the line is not empty is to call the Check Line codeunit to verify that all required journal fields are correct.

Next, the codeunit checks the important table relations. This requires reading the database (by using the **GET** functions). This is why you do it here instead of in **Check Line**.

Before writing to the ledger, the Post Line writes to the register. The first time that the program runs through the Post Line codeunit, it inserts a new record in the **Register** table and sets the **From Entry No.** field to link to the first entry that is posted for the transaction. In every successive run through the Post Line codeunit, the program changes the record by incrementing the **To Entry No**. field.

Then the codeunit takes the next entry number and the values from the journal line and puts them into a ledger record. Finally, it can insert the ledger record.

The last thing that the codeunit does is to increment the variable that holds the next entry number by one. Therefore, when the codeunit is called again, the next entry number is ready.

Post Batch Codeunit

The Post Batch codeunit is responsible for posting all the lines that belong to the same template and batch. Only one record variable for the journal is actually passed to this codeunit. However, the codeunit starts by filtering down to the template and batch of the record that is passed in. Then it determines how many records are in the batch. If there are no records, the codeunit exits without an error. The calling routine then notifies the user that there is nothing to post.

***Note:** The Post Batch codeunit always respects any filters that the user has set on the **Journal Line** table in the **Journal** page. This allows users to only post sections of a batch, instead of the whole batch.*

The Post Batch codeunit can then begin checking each journal line in the batch by calling the Check Line codeunit for each line. As soon as all lines are checked, they can be posted by calling the Post Line codeunit for each line. By then, the codeunit has looped through all the records two times: one time for the Check Line codeunit, and again for the Post Line codeunit.

When the Check Line codeunit checks the validity of a single line, the Post Batch codeunit is responsible for checking the interrelation and consistency of all the lines that are being posted. For example, the Gen. Jnl-Post Batch codeunit also makes sure that the journal lines balance to zero. If a similar check is necessary, it usually occurs as a separate loop through the lines after the Check Line codeunit and before the Post Line loop.

The codeunit may perform other functions, depending on the Journal Template. For recurring journals, the journal lines are updated with new dates based on the date formula. When a recurring journal line is posted, the codeunit must check the **Description** field and the **Document No.** field and replace any parameters with the correct values, for example %1 = day, %2 = week, and %3 = month.

If the template is not recurring, the codeunit deletes all the journal lines after they are successfully posted.

The "Post Batch Process and Data Flow" diagram outlines the steps in the Posting Routine when the Post Batch codeunit is called.

The following diagram shows the logic of a Post Batch codeunit.

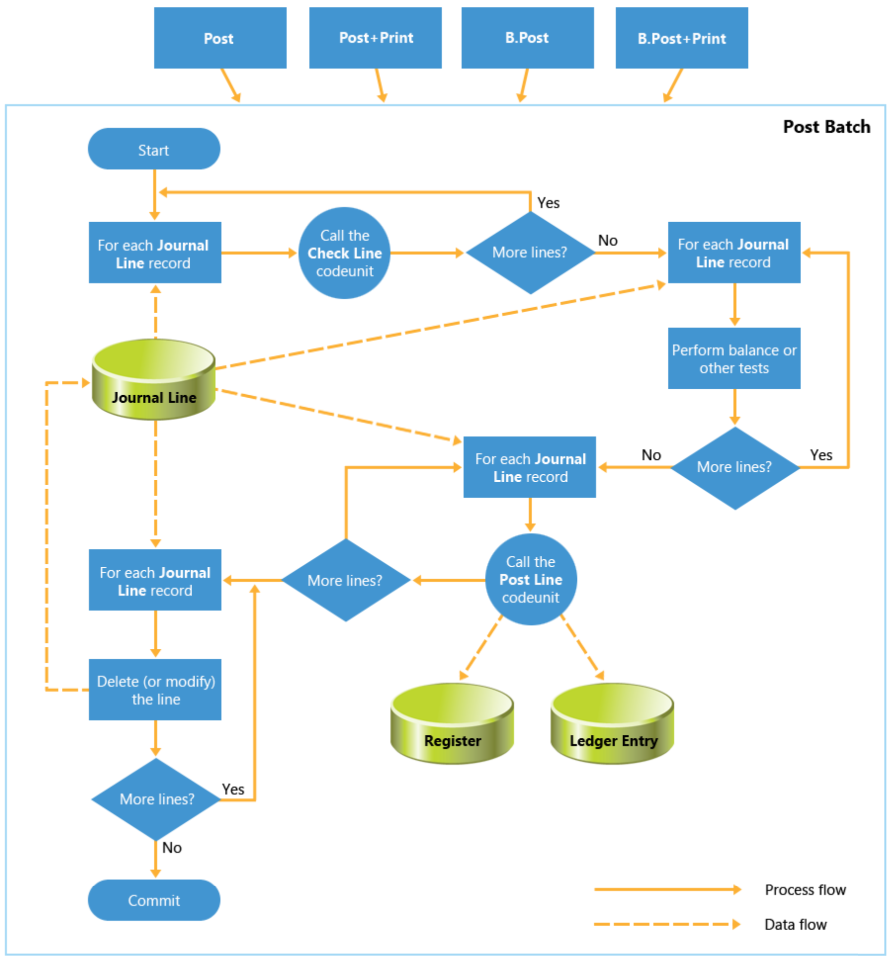

Post
Post+Print
B.Post
B.Post+Print
Post Batch
Start
For each Journal Line record
Call the Check Line codeunit
More lines?
Yes
No
For each Journal Line record
Perform balance or other tests
More lines?
Yes
No
Journal Line
For each Journal Line record
Call the Post Line codeunit
More lines?
Register
Ledger Entry
For each Journal Line record
Delete (or modify) the line
More lines?
Yes
No
Commit
Process flow
Data flow

FIGURE 4.2: POST BATCH PROCESS AND DATA FLOW

Example Posting Routine

For a better understanding of how posting routines are written, you may review one of the simpler posting routines, such as the Resource Journal posting routine.

Check Line

Design the Res. Jnl.-Check Line codeunit (211). Notice that the OnRun trigger gets the **General Ledger Setup** record, and then calls the **RunCheck** function. The **RunCheck** function performs the following required checks:

- If the line is empty, the codeunit skips additional checking and exits without error.
- It checks if the posting date is within the allowed posting date range.
- If the line is related to a time sheet, the **RunCheck** function performs the time sheet checks by calling the Time Sheet Management codeunit.
- It calls functions from the DimensionManagement codeunit to check the dimension combinations and dimension posting rules.

If this codeunit completes without error, then the posting routine continues.

Post Line

Design the Res. Jnl.-Post Line codeunit (212). Notice that the OnRun trigger gets the **General Ledger Setup** record, and then calls the **RunWithCheck** function. The **RunWithCheck** function then calls the **Code** function. Most of the posting work is performed in the **Code** function.

Like the Check Line codeunit, the Post Line codeunit also skips empty lines by exiting. This guarantees that empty lines are not inserted into the ledger. If the line is not empty, it calls **Check Line** to verify that all required journal fields are correct.

Next, the codeunit gets the next entry number from the **Resource Ledger Entry** table to be used with the resource register table. Before writing to the ledger, the Post Line codeunit writes to the register. On the first run, the codeunit adds a new record to the **Register** table. For every successive run through the Post Line codeunit, it increments the **To Entry No** field.

Then the codeunit takes the next entry number and the values from the journal line and puts them into a **Resource Ledger Entry** record. Finally, it inserts the **Resource Ledger Entry** record.

Post Batch

Design the Res. Jnl.-Post Batch codeunit (213). This codeunit is responsible for posting the Resource Template and Resource Batch that is passed to it.

The codeunit starts by filtering to the template and batch of the **Resource Journal Line** record. If no records are found in this range, the Post Batch codeunit exits. The calling routine (codeunit 271, 272, 273, or 274 in this case) notifies the user that there is nothing to post.

The Post Batch codeunit then loops and checks each journal line in the recordset by calling the Check Line codeunit. As soon as all lines are checked, they enter another loop which posts the records by calling the **RunWithCheck** function of the Post Line codeunit for each line.

Unlike the General Journal, there are no interdependencies between Resource Journal lines. Therefore no checks, such as checking the balance must be done.

Finally, this codeunit calls the UpdateAnalysisView codeunit to update any Analysis Views that require updates on posting.

Document Posting Routines

In Microsoft Dynamics NAV 2013, documents provide a simple way to process complex transactions. A document frequently combines multiple transactions into a single transaction. These transactions would be multiple individual transactions if posted from separate journals. By combining these transactions, documents not only simplify work for users, but also guarantee more transactional integrity than journals. This is known as *cross-functional transactional integrity*.

To better understand how a document posting routine works and what its components are, consider the following example of a sales order with three sales lines:

- Line 1: Selling a G/L account – for example, this line may add a surcharge or freight
- Line 2: Selling an item – for example, a computer
- Line 3: Selling a resource – for example, time that an employee spends custom building the computer

When the user posts the document, the program generates an entry that debits the Accounts Receivable Account in the general ledger (G/L). Each document line generates a separate G/L entry for that line. At the same time, the document posting routine generates an entry for the Item and Resource journals, and the General Journal for the Customer.

When these journal entries are posted, they are posted as if the user had entered them into the three separate journals. The biggest difference is that the journal records are posted individually. This enables the Sales-Post routine to bypass the Post Batch codeunit and call the Post Line codeunit directly.

In this example, the document posting routine calls the Gen. Jnl.-Post Line codeunit at least two times: one time for the Item Jnl.-Post Line codeunit, and one time for the Res. Jnl.-Post Line codeunit.

A sales document is posted primarily by the Sales-Post codeunit (80). The whole batch of sales documents can be posted by calling the **Batch Post Sales Invoices** report (297). Be aware that this report is for invoices only. There is a separate report for each document type, such as orders or credit memos.

These reports call the Sales-Post codeunit repeatedly for each document. For this to work, the Sales-Post codeunit must not interact with the user. In fact, the Sales-Post codeunit is never called directly by a page. The page calls the Sales-Post (Yes/No) codeunit (81), the Sales-Post + Print codeunit (82), or one of the reports that was mentioned previously. These other codeunits or reports in turn interact with the user. This is usually to obtain user confirmation before posting, and then to call the Sales-Post codeunit as appropriate.

The “Sales-Post Data Flow “ diagram shows the data flow of the Sales-Post codeunit when a G/L Account, an Item, and a Resource line are included in a sales invoice.

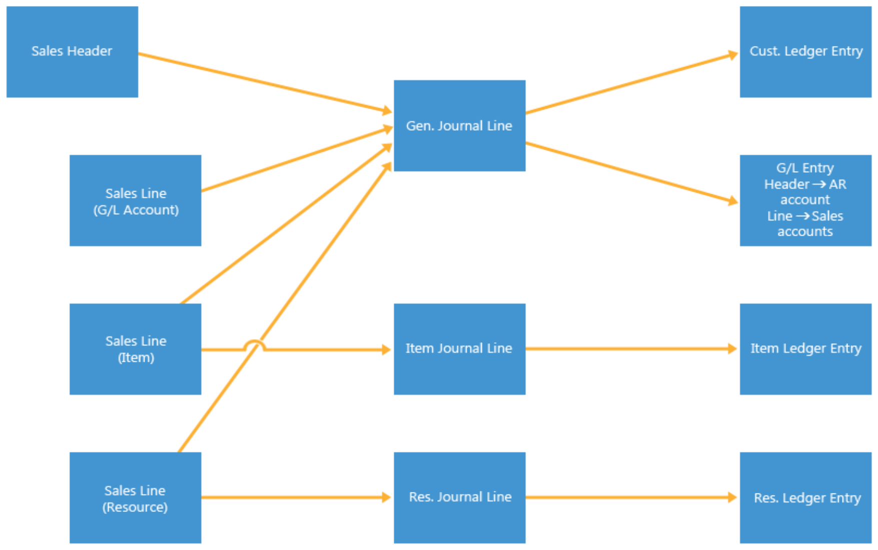


FIGURE 4.3: SALES-POST DATA FLOW

Document Posting Codeunit

Codeunit 80 posts sales documents. When a user ships and invoices a sales order, much of the work is performed in codeunit 80. This codeunit performs the following tasks:

- Determines the document type and validates the information on the sales header and lines.
  - The codeunit determines the posted document numbers and updates the header.
  - This section ends with a **COMMIT** function call.

- Locks the appropriate tables.
- Inserts the **Shipment Header** and **Invoice** or **Credit Memo Header**.
- Clears the **InvPostingBuffer**, a temporary table that is based on the **Invoice Post. Buffer** table.
- Within the main **REPEAT** loop, it iterates through all sales line, and checks each line with its matching **Sales Shipment Line** (if the line is previously shipped).
  - If the line type is Item or Resource, it is posted through the appropriate journal. The line is then added to the posting buffer. When you add to the posting buffer, a new line may be inserted, or you can update a current line. The buffer makes sure that there are as few **G/L Entry** records as possible that result from a single transaction. Therefore, it always combines lines that have the same values for the **G/L Account No.** field, the same dimension values, the same posting groups, and some more important fields.
  - If the line is related to a job, codeunit 80 posts a journal line through the Job Journal.
  - If there is no shipment line, codeunit 80 inserts one.
  - Finally, codeunit 80 copies the **Sales Line** to the **Invoice Line** or **Credit Memo Line** (the posted tables).

- Posts all entries in the **Posting Buffer** temporary table to the **General Ledger** table.
  - These are the **Credits** that are created from the sale of the lines.
  - Then the codeunit can post the **Debit** to the **General Ledger**.

  - The customer entry is made to the **Sales Receivables Account**.
  - The routine then checks whether there is a balancing account for the header. This corresponds to an automatic payment for the invoice.

- Updates and deletes the **Sales Header** and **Lines** and commits all changes.

Documentation in Existing Objects

When you make changes to an existing object, enter a note in the Documentation trigger, in the same manner as for new objects that were discussed in earlier modules. The note should contain a reference number, the date the modification is completed, the name of the developer who made the modification, and a short description of the change.

Following is an example taken from the **Resource** table, with notes from Module 2, Lab 1.

Documentation Trigger Example

```
CSD1.00 - 2012-06-15 - D. E. Veloper

  Module 2 - Lab 1

    - Added new fields:

      - Internal/External

      - Maximum Participants
```

Notice that documentation in an existing object resembles the documentation in a new object. The details of these notes and the way that they are formatted can vary from one developer to another. They may also be the subject of an organization-wide set of standards. Understand that these notes are necessary to keep track of the changes that are made to objects over the life span of the objects in a Microsoft Dynamics NAV 2013 application.

Code Comments

Together with the general comments that are provided in the Documentation trigger, it is important to provide comments in the code at the lines where a change is made. Do this only when changing an existing object, not when you create a new object.

The key is to mark the changed code with the same reference number as used in the Documentation trigger of the object.

For example, mark a single changed line of code as follows.

Single Line Modification

```
//CSD1.00>

//CheckCustBlockage("Sell-to Customer No.",TRUE);

CheckCustBlockage("Sell-to Customer No.",FALSE);

//CSD1.00<
```

If you add or change a whole block of code, mark the change as follows.

Multiple Lines Modification

```
//CSD1.00>

//TESTFIELD("Document Type");

//TESTFIELD("Sell-to Customer No.");

//TESTFIELD("Bill-to Customer No.");

//TESTFIELD("Posting Date");

//TESTFIELD("Document Date");

CheckDocumentTypeFields("Document Type");

IF "Sell-to Customer No." <> "Bill-to Customer No." THEN

  CompareCustomerDimensions("Sell-to Customer No.","Bill-to Customer No.");

//CSD1.00<
```

When removing standard code, mark the removed lines as follows.

Code removal

```
//CSD1.00>

//SalesLine.SETFILTER(Quantity,'<>0');

//SalesLine.SETFILTER("Return Qty. to Receive",'<>0');

//CSD1.00<
```

Never delete the original Microsoft Dynamics NAV 2013 code. The goal of code comments is to preserve the original code, even when you change the business logic or remove the business logic. Preserving the original code performs the following three important tasks:

- It shows the original code before any customization.
- It makes any changes more obvious in the source code of the object.
- It streamlines the upgrade process by enabling the upgrade tools to match the original code with the new version code.

Note: *Even though braces are a valid way to comment out multiple lines of code, use them sparingly. Comments made with braces are not color-coded in green, and are inconspicuous when you are viewing the code. Also, during upgrade projects they make the changes less obvious than the changes that you make with // comments.*

Performance Issues

When you write large posting routines, it is important to program to maximize performance. There are several steps to program a solution in Microsoft Dynamics NAV that will improve performance.

Table Locking

Most of the time, you do not have to be concerned about transactions and table locking when you develop applications in Microsoft Dynamics NAV Development Environment, because the SQL Server adds necessary locks to affected tables as soon as you start inserting, changing, or deleting data. However, there are some situations when you must explicitly lock a table to guarantee process or transaction integrity.

For example, suppose that in the beginning of a function, you read the data from a table, and then later use that data to perform various checks and calculations. Then, you write the record back to the database, based on the result of this processing. The values that you retrieved at the beginning must be consistent with the final data in the table. In short, other users must be unable to update the table while a function is busy doing the calculations.

The solution is to lock the table at the beginning of the function by using the **LOCKTABLE** function. This function locks the table until the write transaction is committed or rolled back. This means that other users can read from the table. However, they cannot write to it. Calling the **COMMIT** function unlocks the table.

Note: *The **LOCKTABLE** function does not necessarily lock the table. The SQL Server may decide to lock only the rows that you read, or to escalate the locks to greater levels, such as a page or even a table level. Regardless of whether the SQL Server locks the table or only the sections of it, the **LOCKTABLE** function guarantees data consistency until you either call the **COMMIT** function or roll back the transaction.*

Reducing Impact on the Server

Good code design minimizes the load on the server. There are several ways to achieve this including the following:

- Try to use the **COMMIT** function as little as possible. This function is handled automatically by the database for most circumstances.
- Use the **LOCKTABLE** function only when it is necessary. Remember that an insert, change, rename, or delete function automatically locks the table. So for most situations you do not have to lock the tables.
- Structure the code so that you do not have to examine the return values of the **INSERT**, **MODIFY**, or **DELETE** functions. When you use the return value, the server must be notified immediately to obtain a response. Therefore, if they are not necessary, do not examine the return values of **INSERT**, **MODIFY**, or **DELETE** functions.
- Use the **CALCSUMS** and **CALCFIELDS** functions when possible to avoid examining records to total values. Use the **SETAUTOCALCFIELDS** function when you must obtain the value of a FlowField for every single row in a loop.
- Always avoid too many round trips to the server.

Reducing Impact on Network Traffic

Consider setting keys and filters, and then use **MODIFYALL** or **DELETEALL** functions. These functions send only one command to the server, instead of getting and deleting or changing many records one by one using the **MODIFY** and **DELETE** functions.

Because **CALCSUMS** and **CALCFIELDS** can both take multiple parameters, you can use these functions to perform calculations on several fields that have a single function call.

Posting Seminar Registrations

In this section the client's functional requirements are analyzed and a solution is designed and implemented.

Solution Design

The CRONUS International Ltd. functional requirements outline that when a seminar is completed, users must be able to move the seminar registration information into the transaction history, and disable any further modification of this information. This requirement indicates that there must be a posting process that is involved with seminar registrations. When you apply the customer’s language to the terminology of Microsoft Dynamics NAV 2013, this requirement states that users must be able to post the seminar registration information.

Another requirement further clarifies the customer’s business need for a transaction history for seminars that must include the following:

- Details of participants, instructors, and rooms that are utilized during the seminars
- Information about additional charges

This information will be the basis for seminar cost analytical and statistical reporting. This indicates that the detailed information about posted transactions must include all the information that is contained in the seminar registration document. Microsoft Dynamics NAV terminology calls these transactions the *Ledger Entries*.

The final requirement details how the seminar registration information must integrate with the availability planning functionality for instructors and rooms. It also must provide the basis for automatic invoicing of customers.

When seminar registration is posted, the resource ledger entries should be generated for the instructor and room resources. The solution must provide the registration posting functionality that creates transaction data from which users can view history, analyze statistics, and create invoices.

When you introduce posting functionality, you must follow Microsoft Dynamics NAV 2013 standard conceptual, data model, and user interface principles. This means that you must provide at least the journal posting infrastructure that consists of the following:

- Journal tables
- Ledger tables
- Register tables
- Journal posting codeunits

Depending on the complexity of the processes that you must support, you may have to extend the functionality to also include the following features:

- Posted document tables
- Document posting codeunits

The "Data Flow in Seminar Registration Posting" figure shows the entities that are involved in the seminar registration posting process and the data flow between them.

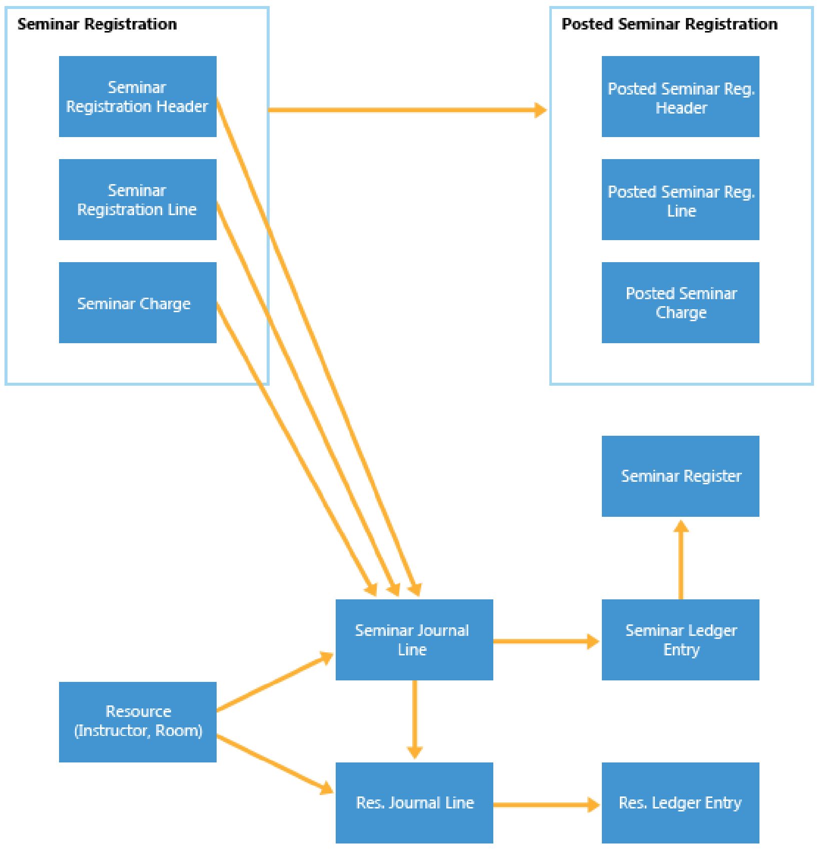


FIGURE 4.4: DATA FLOW IN SEMINAR REGISTRATION POSTING

Development

You must develop the tables, pages, and codeunits to enable the seminar registration posting process and keep the transaction history. All tables, pages, and codeunits must follow the standard Microsoft Dynamics NAV 2013 principles, and must provide all functionality that users experience with other posting routines and transactional history features.

Tables

To support the posting process and to keep the transaction history for the Seminar Management module, you must create the following new tables.

Table	Remarks
123456718 Posted Seminar Reg. Header	Holds the information for the completed (posted) seminar. Takes the data from the **Seminar Registration Header** table during posting.
123456719 Posted Seminar Reg. Line	Holds the detailed information for the completed (posted) seminar. Takes the data from the **Seminar Registration Line** table during posting.
123456721 Posted Seminar Charge	Holds charges that are related to the completed (posted) seminar.
123456731 Seminar Journal Line	Lets you post the seminar ledger entries.
123456732 Seminar Ledger Entry	Keeps the transaction details for all posted seminars.
123456733 Seminar Register	Keeps the transaction log for posted seminars.

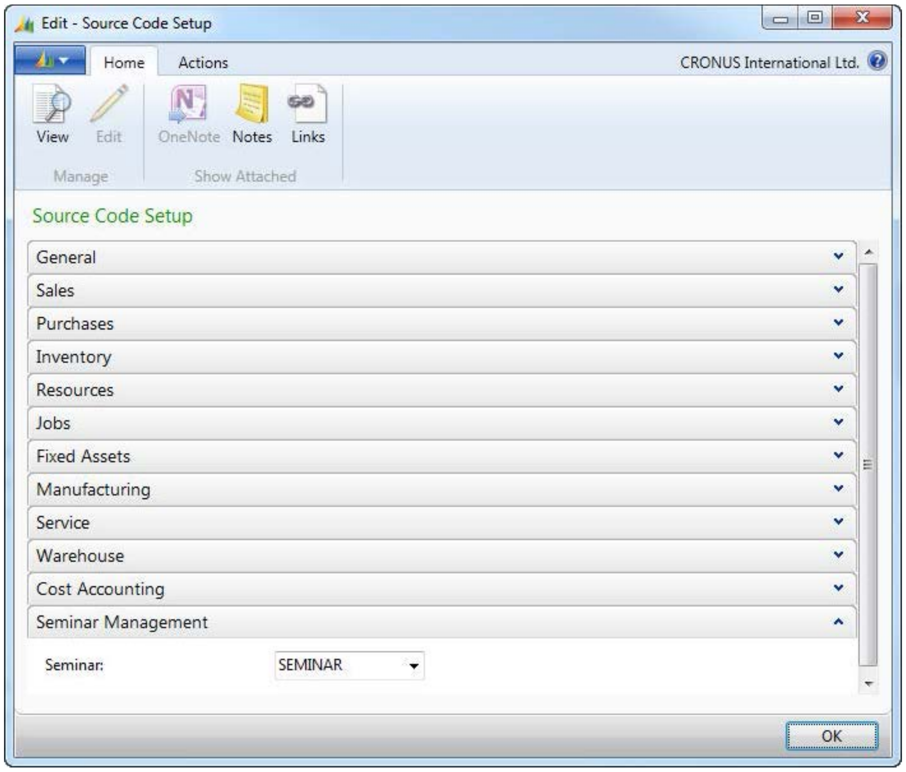
You must also change the following tables.

Table	Remarks
203 Res. Ledger Entry	Add fields to link the resource ledger entries to the seminars and to the posted seminar registration documents.
207 Res. Journal Line	Add fields to support posting of seminar information during resource journal posting.
242 Source Code Setup	Add a field to support the audit trail source code for the seminar registration transaction history.

Pages

The pages for the seminar registration posting and the navigation between them reflect the relationships that are shown in the "Data Flow in Seminar Registration Posting" diagram. Design the simplest pages first and then integrate them with the more complex pages.

Add the Seminar Management group and the **Seminar** field to the **Source Code Setup** page:



Seminar: SEMINAR

FIGURE 4.5: THE SOURCE CODE SETUP PAGE (279)

The **Seminar Ledger Entries** page displays the ledger entries:

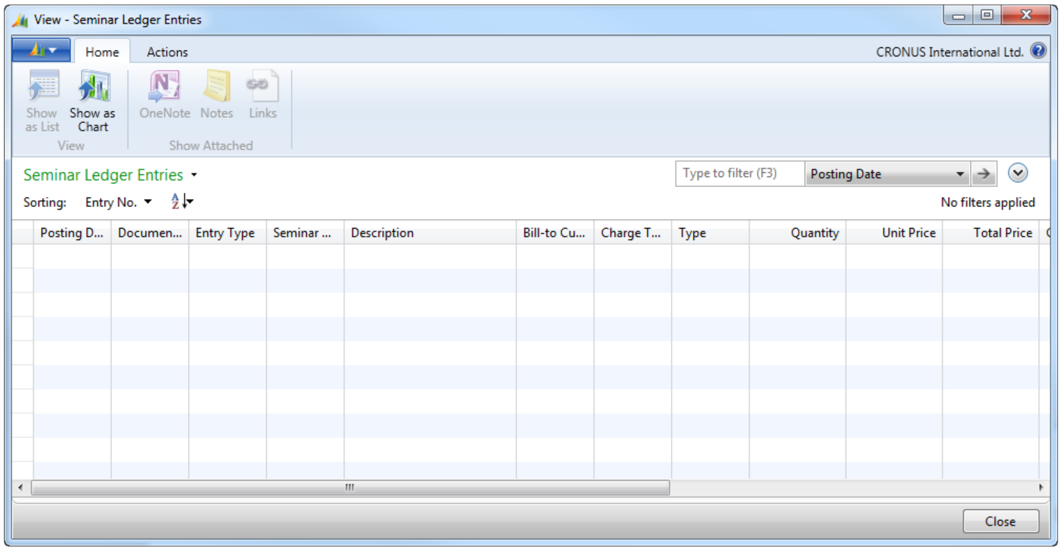


FIGURE 4.6: THE SEMINAR LEDGER ENTRIES PAGE (123456721)

The “Seminar Registers Page” figure displays the registers that are created when seminar registrations are posted.

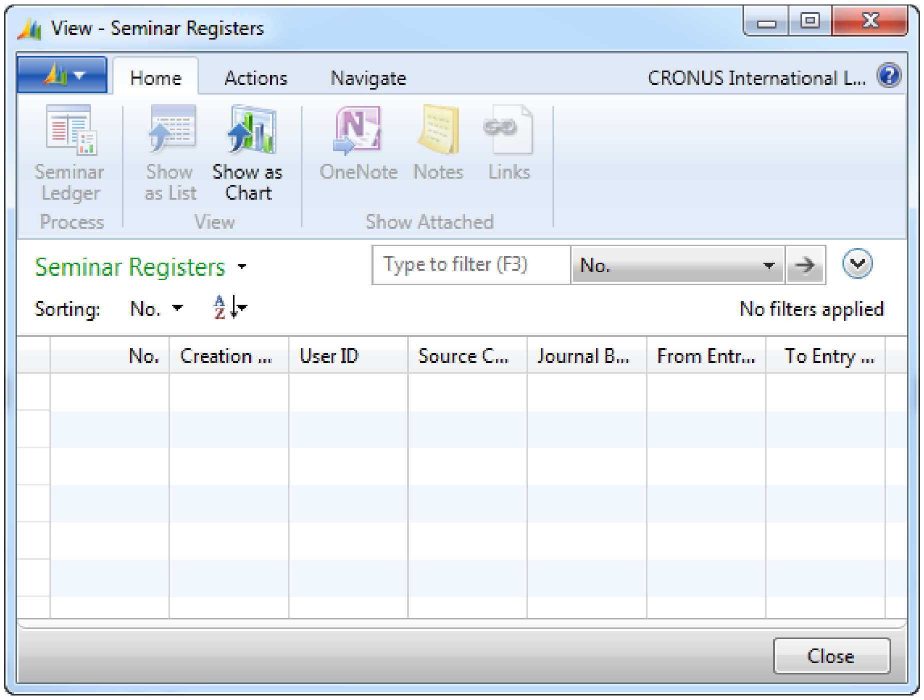


FIGURE 4.7: THE SEMINAR REGISTERS PAGE (123456722)

The "Posted Seminar Charges Page" image shows the charges that are related to a posted seminar registration.

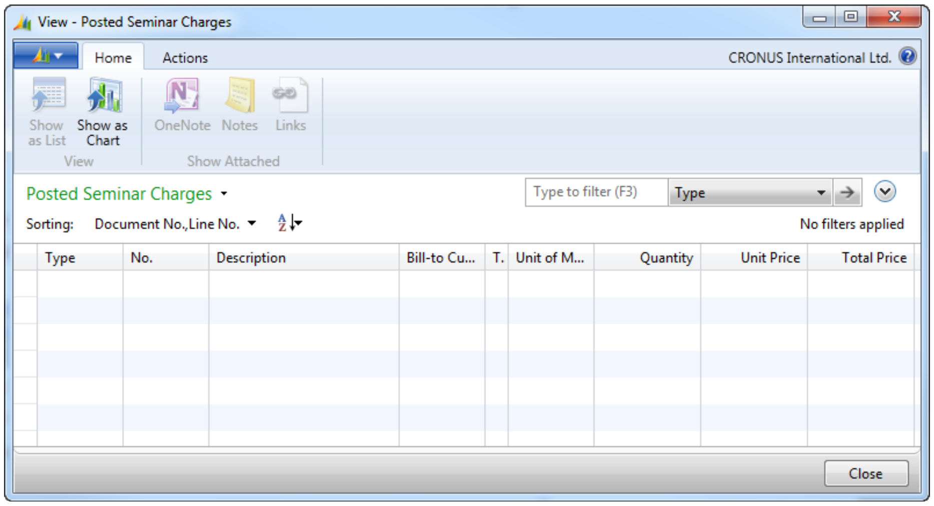


FIGURE 4.8: THE POSTED SEMINAR CHARGES PAGE (123456739)

The "Posted Seminar Reg. Subform page" image shows the lines page for the **Posted Seminar Registration** document page.

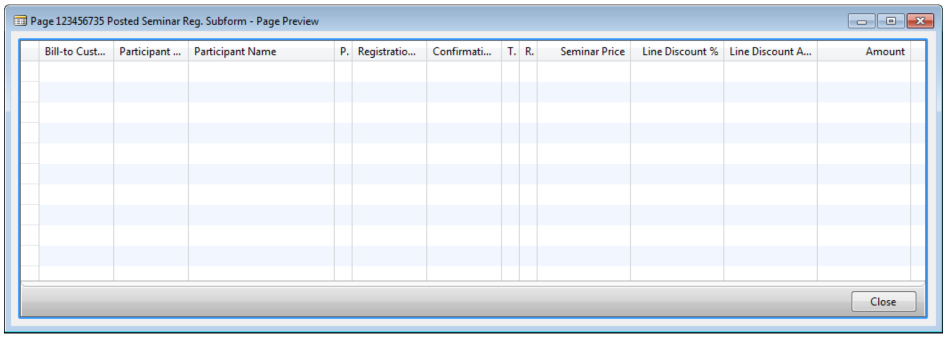
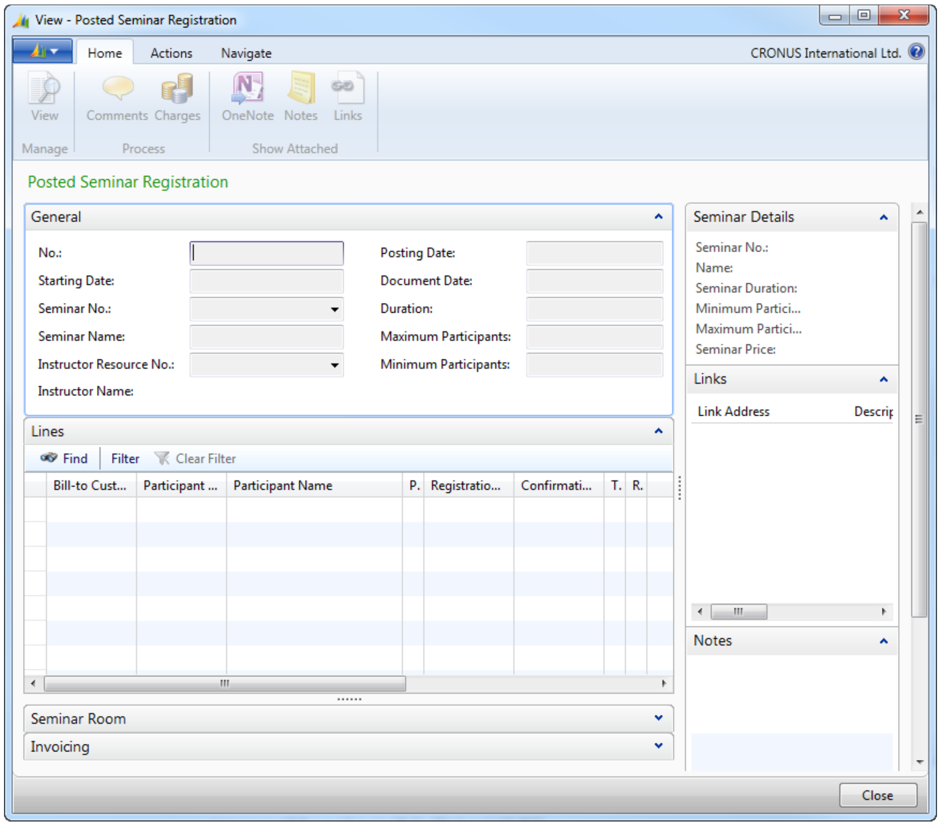


FIGURE 4.9: THE POSTED SEMINAR REG. SUBFORM PAGE (123456735) IN PAGE PREVIEW

The "Posted Seminar Registration Page" image shows the posted seminar registration document.



Posted Seminar Registration

General

No.:		Posting Date:	
Starting Date:		Document Date:	
Seminar No.:		Duration:	
Seminar Name:		Maximum Participants:	
Instructor Resource No.:		Minimum Participants:	
Instructor Name:			

Lines

Bill-to Cust...	Participant ...	Participant Name	P.	Registratio...	Confirmati...	T.	R.

Seminar Room

Invoicing

Seminar Details

Seminar No.:
Name:
Seminar Duration:
Minimum Partici...
Maximum Partici...
Seminar Price:

Links

Link Address	Descrip

Notes

FIGURE 4.10: THE POSTED SEMINAR REGISTRATION PAGE (123456734)

The “Posted Seminar Reg. List Page” figure displays a list of posted seminar registrations.

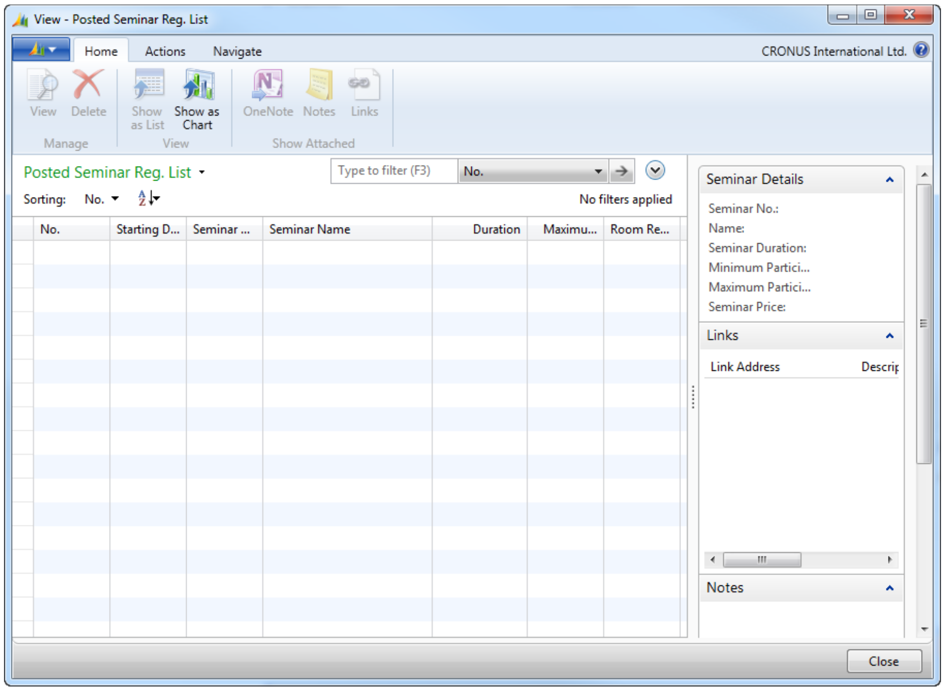


FIGURE 4.11: THE POSTED SEMINAR REG. LIST PAGE (123456736)

Codeunits

As in all journal postings, the journal posting codeunits check Seminar Journal lines and post them. However, unlike some posting codeunits (such as General Journal), a codeunit that posts a batch of these journal lines is not required because posting batches is not required for this solution.

You must develop the following journal posting codeunits.

Codeunit	Remarks
123456731 Seminar Jnl.-Check Line	• Verifies the data validity of a seminar journal line before the posting routine posts it by doing the following: • The codeunit checks that the journal line is not empty and that there are values for the **Posting Date**, **Instructor Resource No.**, and **Seminar No.** fields. • Depending on whether the line is posting an Instructor, a Room, or a Participant, the codeunit checks that the applicable fields are not blank. • The codeunit also verifies that the dates are valid.

Codeunit	Remarks
123456732 Seminar Jnl.-Post Line	Performs the posting of the **Seminar Journal Line**. The codeunit creates a **Seminar Ledger Entry** for each **Seminar Journal Line** and creates a **Seminar Register** to track which entries are created during the posting

Change the Res. Jnl.-Post Line codeunit to make sure that the **Seminar No.**, and the **Seminar Registration No.** fields are recorded in the **Res. Ledger Entry** table.

Finally, you must develop the codeunits for the seminar registration document posting.

Codeunit	Remarks
123456700 Seminar-Post	Posts the complete seminar registration that includes the resource posting and seminar posting. • The codeunit transfers the comment records to new comment records that correspond to the posted document. The codeunit also copies charges to new tables that contain posted charges. • The codeunit creates a new Posted Seminar Reg. Header record and **Posted Seminar Reg. Line** records. • The codeunit then runs the job journal posting, and posts seminar ledger entries for each participant, for the instructor, and for the room. • Finally, the codeunit deletes the records from the document tables. This includes the header, lines, comment lines, and charges.
123456701 Seminar-Post (Yes/No)	Interacts with users, and confirms that they want to post the registration. If users confirm the posting, the codeunit runs the Seminar-Post codeunit.

Lab 4.1: Reviewing and Completing the Journal and Ledger Tables

Scenario

Isaac is a junior developer working on the team that is implementing Microsoft Dynamics NAV 2013 for CRONUS International Ltd. Isaac is in charge of developing the base version of tables and user interface objects.

When he developed the seminar registration tables and pages, he only provided you with the text file. This made it more difficult to review the contents of the input file, because you had to review the text file by using a text-editing tool, such as Notepad. You advised Isaac to always provide both the text and .fob versions of the object import file to make the review process simpler.

Isaac created the seminar journal and ledger tables and pages, and delivered both the .fob and text files that contain the objects. As Isaac's supervisor, you now must review the objects to make sure that they follow all Microsoft Dynamics NAV 2013 architectural principles and best practices. You must make any necessary corrections to the tables, their properties and fields, and their code.

By this point, you should already be familiar with the basics of the Microsoft Dynamics NAV 2013 Development Environment functions and features. Therefore, the detailed instructions only give you descriptions of the actions that you must do, instead of giving you detailed steps that are suitable for beginning users.

The following table contains some examples.

Instead of...	... the instructions state
1. On the **View** menu, click **Properties**. 2. In the Editable property, enter "No". 3. Close the **Properties** window.	1. Set the Editable property to No.
1. On the **View** menu, click **C/AL Locals**. 2. On the **Return Value** tab, in the **Return Type** field, enter "Boolean". 3. Close **C/AL Locals**.	1. Set the **Return Value** for the function to return Boolean type.
1. In Object Designer, click **All**.	1. View all objects in Object Designer.

Instead of...	... the instructions state
1. In Object Designer, click **Page**. 2. Find and select the page 123456721, **Seminar Ledger Entries**. 3. Click **Design**.	1. Design the page 123456721, **Seminar Ledger Entries**.
1. On the **File** menu, click **Save**. 2. In the **Save** dialog window, make sure that the **Compiled** check box is selected, and then click **OK**. 3. Close the table.	1. Compile, save, and then close the table.

Note: *This applies only to the concepts that were covered earlier. If there is a new concept, the full detailed steps are provided. For any repeated concepts, only the descriptive instructions are provided. If you are still unsure about the detailed steps of a task, refer to the labs in earlier modules. If you have any questions, ask your instructor.*

Exercise 1: Reviewing the Import File Contents and Importing the Objects

Exercise Scenario

Prior to importing objects, you should review the contents of the import file to make sure that no existing objects will be overwritten. Start the review process by importing a .fob file. This enables you to review the contents of the file before saving objects to the database.

Note: *You may notice that the import file does not include the* ***Seminar Journal Template*** *or* ***Seminar Journal Batch*** *table, or the* ***Seminar Journal*** *page. This is because the* ***Seminar Journal Template*** *is always posted in the background as a part of the document posting routine. For document posting, the template and the batch are always undefined. Therefore the tables are not needed.*

Provide only the ***Seminar Journal Template*** *and* ***Seminar Journal Batch*** *tables. Provide a* ***Journal*** *page if users must access the journal directly from the client.*

Task 1: Preview the .fob File Contents

High Level Steps

1. In Object Designer, open the Mod04\Labfiles\Lab 4.A - Starter.fob file in the Import Worksheet.
2. Make sure that all listed objects are new.
3. Close the Import Worksheet.

Detailed Steps

1. In Object Designer, open the Mod04\Labfiles\Lab 4.A - Starter.fob file in the Import Worksheet.
    a. Open **Object Designer**.
    b. On the **File** menu, click **Import**.
    c. In the **Import Objects** dialog box, browse to Mod04\Labfiles\Lab 4.A - Starter.fob.
    d. Click **Open**. The confirmation dialog box opens, and notifies you that no conflicts were found.
    e. Click **No**, to open the Import Worksheet, as instructed by the dialog window.

2. Make sure that all listed objects are new.
    a. Check the Action column for each row.
    b. Make sure that the Action is set to Create for each row.

3. Close the Import Worksheet.
    a. Click **Cancel** to close the Import Worksheet.

Note: *Even though you could click* ***OK*** *to complete the import, you want to import the text file to review only the new objects and compile them manually. When you import from a text file, any new objects are obvious because they are not compiled.*

Task 2: Import and Compile the Objects

High Level Steps

1. In Object Designer, import the Mod04\Labfiles\Lab 4.A - Starter.txt file.
2. Select and compile the imported objects.

Detailed Steps

1. In Object Designer, import the Mod04\Labfiles\Lab 4.A - Starter.txt file.
   a. In Object Designer, on the **File** menu, click **Import**.
   b. In the Import Objects dialog box, browse to Mod04\Labfiles\Lab 4.A - Starter.txt file.
   c. Click **Open**.

2. Select and compile the imported objects.
   a. View all objects in **Object Designer**.
   b. Filter the view to only show the uncompiled objects.
   c. Compile the objects.

Exercise 2: Reviewing the Seminar Journal Line Table

Exercise Scenario

After importing the objects, you must make sure that all objects follow the best practices and standard Microsoft Dynamics NAV 2013 principles. Start with the **Seminar Journal Line** table.

Task 1: Review Table and Field Properties

High Level Steps

1. Clear all filters in Object Designer.
2. Make sure that all important standard journal fields are present in the table 123456731, **Seminar Journal Line** table, and that the primary key consists of the **Journal Template Name**, **Journal Batch Name** and **Line No.** fields in the correct order.

Detailed Steps

1. Clear all filters in Object Designer.
   a. In Object Designer, click **View** > **Show All**, or press SHIFT+CTRL +F7.

2. Make sure that all important standard journal fields are present in the table 123456731, **Seminar Journal Line** table, and that the primary key consists of the **Journal Template Name**, **Journal Batch Name** and **Line No.** fields in the correct order.
   a. Design the table 123456731, **Seminar Journal Line**.

b. Make sure that the following fields are present in the table.

Field	Type	Remarks
Journal Template Name	Code[10]	
Journal Batch Name	Code[10]	This field must exist in all journal line tables, but is frequently located after all the fields that describe the transaction.
Posting Date	Date	Specifies the date for the entry. This is the date that the transaction occurred.
Document Date	Date	Specifies the date for the document. By default this equals the Posting Date. Users can change this date because the transaction may be entered and posted on different dates.
Source Code	Code[10]	Specifies the source for the entry. Sources map directly to journal templates. Therefore, they all map to transaction types. This field forms the basis for the audit trail that Microsoft Dynamics NAV 2013 leaves for every transaction. Users cannot change this field.
Reason Code	Code[10]	Specifies the reason why the entry was posted. Users may change this field.

Note: *There are many more fields in the table. Most of the other fields are specific to the seminar journal transactions.*

c. On the **View** menu, click **Keys**.

d. Verify that the first key in the list is "Journal Template Name,Journal Batch Name,Line No."

e. Close the **Keys** window.

Task 2: Review Table Code

High Level Steps

1. Check whether the table contains the necessary functions for the journal line tables.
2. Create and define the **EmptyLine** function. The function must return a Boolean value.
3. Enter the code that sets the **Document Date** field to the value of the **Posting Date** field when users edit the **Posting Date** field. Then save and close the table.

Detailed Steps

1. Check whether the table contains the necessary functions for the journal line tables.
   a. In the **C/AL Globals** window, check the **Functions** tab to determine whether there are any functions that were defined.

Note: *All **Journal Line** tables must contain the **EmptyLine** function, which is called during posting to make sure that only non-empty lines are posted.*

2. Create and define the **EmptyLine** function. The function must return a Boolean value.
   a. Create the **EmptyLine** function, configure its **Return Value** to return Boolean type.
   b. Open the **C/AL Editor** window to view the code for the table.
   c. In the **EmptyLine** function trigger, enter the following code.

```
EXIT(

  ("Seminar No." = '')

  AND (Quantity = 0));
```

Note: *The number of conditions in the **EmptyLine** function depends on the complexity of the journal transaction. For the seminar journal, if both the **Seminar No.** and **Quantity** fields are undefined, then the journal line is considered empty.*

3. Enter the code that sets the **Document Date** field to the value of the **Posting Date** field when users edit the **Posting Date** field. Then save and close the table.
   a. In the Posting Date – OnValidate trigger, enter the following code.

```
VALIDATE("Document Date","Posting Date");
```

   b. Compile, save, and then close the table.

Exercise 3: Reviewing Other Tables

Exercise Scenario

After reviewing the **Seminar Journal Line** table and correcting the issues that caused the gap between Isaac's work and Microsoft Dynamics NAV 2013 standards and best practices, you want to review the remaining journal posting tables: the **Seminar Ledger Entry** and **Seminar Register** tables.

Task 1: Review the Seminar Ledger Entry Table

High Level Steps

1. Review all the fields in the table 123456732, **Seminar Ledger Entry** to note any differences between the actual **Seminar Ledger Entry** table fields and the required set of fields.
2. Correct the issues that were noted in the previous step and save the table.

Detailed Steps

1. Review all the fields in the table 123456732, **Seminar Ledger Entry** to note any differences between the actual **Seminar Ledger Entry** table fields and the required set of fields.
   a. Design the table 123456732, **Seminar Ledger Entry**.
   b. Compare the fields in the table with the following list.

Field No.	Field Name	Data Type	Length
1	Entry No.	Integer	
2	Seminar No.	Code	20
3	Posting Date	Date	
4	Document Date	Date	
5	Entry Type	Option	
6	Document No.	Code	20
7	Description	Text	50
8	Bill-to Customer No.	Code	20

Field No.	Field Name	Data Type	Length
9	Charge Type	Option	
10	Type	Option	
11	Quantity	Decimal	
12	Unit Price	Decimal	
13	Total Price	Decimal	
14	Participant Contact No.	Code	20
15	Participant Name	Text	50
16	Chargeable	Boolean	
17	Room Resource No.	Code	20
18	Instructor Resource No.	Code	20
19	Starting Date	Date	
20	Seminar Registration No.	Code	20
23	Res. Ledger Entry No.	Integer	
24	Source Type	Option	
25	Source No.	Code	20
26	Journal Batch Name	Code	10
27	Source Code	Code	10
28	Reason Code	Code	10
29	No. Series	Code	10
30	User ID	Code	20

Note: *Notice that the actual set of fields does not match this table. Apparently, Isaac has only created the* ***Seminar Ledger Entry*** *table by copying the* ***Seminar Journal Line*** *table. You now must correct these issues.*

c. Note the following differences.

Issue	Remarks
Entry No.	This field does not exist. Instead, there are the **Journal Template Name** and **Line No.** fields.
No. Series	This field does not exist. Instead, there is the **Posting No. Series** field.
User ID	This field does not exist.

Issue	Remarks
Primary key	Primary key must only include the **Entry No.** field.

2. Correct the issues that were noted in the previous step and save the table.
   a. From the list of fields, delete the **Journal Template Name** and **Line No.** fields.
   b. Create the **Entry No.** field as the first field in the table.
   c. Create the **User ID** field as the last field in the table.
   d. Compile, save, and close the table.
   e. Design the **Seminar Ledger Entry Table** again.

Note: *When you add a new field to a table, you must close the table and then reopen it before you can reference the field in the code.*

   f. Rename the **Posting No. Series** field to **No. Series**.
   g. Set the Caption properties for any fields that you created or changed to match the field's Name.
   h. Set the TableRelation property of the **User ID** field to relate to the **User Name** field of the table **User**.

Note: *If you are unsure how to set the TableRelation property directly, click the **AssistEdit** button for **TableRelation**, and establish the relation in the **Table Relation** window.*

   i. Set the ValidateTableRelation and TestTableRelation properties of the **User ID** field to **No**.

Note: *This guarantees that if the user is deleted from the database later, no table relation tests fail.*

   j. Set the primary key to **Entry No.**
   k. In the OnValidate trigger for the **User ID** field, enter the code that calls the **LookupUserID** function of the User Management codeunit.

```
UserMgt.LookupUserID("User ID");
```

Note: *Make sure that you define the* UserMgt *local variable for the User Management codeunit.*

l. Renumber the **Field No.** for all the fields incrementally from 1 to 28, starting with the **Entry No.** field and ending with the **User ID** field.

m. Compile, save, and then close the table.

Task 2: Review the Seminar Register Table

High Level Steps

1. Review all the fields in the table 123456733, **Seminar Register** to note any differences between the actual **Seminar Register** table fields and the required set of fields.
2. Correct the issues that you noted in the previous step and save the table.

Detailed Steps

1. Review all the fields in the table 123456733, **Seminar Register** to note any differences between the actual **Seminar Register** table fields and the required set of fields.
   a. Design the table 123456733, **Seminar Register**.
   b. Compare the fields in the table with the following list:

Field No.	Field Name	Data Type	Length
1	No.	Integer	
2	From Entry No.	Integer	
3	To Entry No.	Integer	
4	Creation Date	Date	
5	Source Code	Code	10
6	User ID	Code	20
7	Journal Batch Name	Code	10

   c. Note the following issues:

Issue	Remarks
No.	The field does not exist. Instead, the field **Entry No.** is there. By convention, **Register** tables always have the primary key field named **No.**
Journal Template Name	This field is not necessary in the register. The source of the transaction is kept in the **Source Code** field.

2. Correct the issues that you noted in the previous step and save the table.
    a. Change the Name and the Caption properties for the **Entry No.** field to "No."
    b. Delete the **Journal Template Name** field.
    c. Renumber the **Field No.** for all the fields incrementally from 1 to 7, starting with the **No.** field and ending with the **Journal Batch Name** field.
    d. Compile, save, and then close the table.

Exercise 4: Customize the Source Code Setup Table and Page

Exercise Scenario

In Microsoft Dynamics NAV 2013 every transaction must leave a clear and obvious audit trail. The core feature for auditing transactions in Microsoft Dynamics NAV 2013 is the source codes feature. Source codes simplify locating transactions that originated from a specific application function, such as a journal or a batch job.

When customizing Microsoft Dynamics NAV 2013 to include new posting routines or batch jobs that result in posted entries, you must extend the **Source Code Setup** table and page by using a field that identifies any new transaction type that you are introducing. Then in your posting routines, you must make sure that you use that field to identify the transactions that originated from that new feature. Therefore, you establish an audit trail that is consistent with other features of Microsoft Dynamics NAV 2013.

Note: *Transaction source codes can only be defined in the* ***Journal Template*** *tables for journals, and in the* ***Source Code Setup*** *table for documents and batch jobs. Every transaction carries the* ***Source Code*** *field, and this field is always taken from either the appropriate* ***Journal Template*** *table, or from the* ***Source Code Setup*** *table. Users can never change the* ***Source Code*** *field in any of the transactions. The* ***Source Code*** *field is only shown in the* ***Ledger Entries*** *and* ***Register*** *pages, never in journals or documents.*

After reviewing all posting tables, you now must make sure that the **Source Code Setup** table includes a source code configuration field for the seminar posting routine. Then, you must add the same field to the **Source Code Setup** page.

Task 1: Customize the Source Code Setup Table

High Level Steps

1. Add the **Seminar** field to the table 242, **Source Code Setup**.

Detailed Steps

1. Add the **Seminar** field to the table 242, **Source Code Setup**.
   a. Design the table 242, **Source Code Setup**.
   b. Add the following field.

No.	Field Name	Type	Length	Remarks
123456700	Seminar	Code	10	Set the table relation to the **Source Code** table.

   c. Set the Caption property for the field to "Seminar".
   d. Compile, save, and then close the table.

Task 2: Customize the Source Code Setup Page

High Level Steps

1. Add the **Seminar Management** FastTab and the **Seminar** field to the page 279, **Source Code Setup**.

Detailed Steps

1. Add the **Seminar Management** FastTab and the **Seminar** field to the page 279, **Source Code Setup**.
   a. Design the page 279, **Source Code Setup**.
   b. Add the Seminar Management group control as the last group control, just before the FactBoxArea container. Make sure that you indent the group control at the same level as the other group controls.
   c. Add the **Seminar** field to the Seminar Management group. Make sure that it is indented under the group.
   d. Compile, save, and then close the page.

Lab 4.2: Creating Codeunits and Pages for Seminar Journal Posting

Scenario

After you have reviewed all journal posting tables, and have completed the necessary customizations and corrections, you are now ready to develop the codeunits for the seminar journal posting routine. CRONUS International Ltd. only requires document posting functionality, and did not request journal posting functionality. Their users would find it unnatural to post any seminar registrations through a journal. Therefore, you do not have to develop all journal posting codeunits that you would normally provide if the customer required a full journal user interface.

Note: *Even though the customer never requested it, you must provide the journal posting functionality to enable documents to post ledger entries in a way that is consistent with both best practices and the application standards that are found in other functional areas.*

The journal posting codeunits that you must develop are as follows.

Codeunit	Remarks
Seminar Jnl.-Check Line	This codeunit checks each line before posting. You must always provide this codeunit.
Seminar Jnl.-Post Line	This codeunit posts each line. You must always provide this codeunit.
Seminar Reg.-Show Ledger	This codeunit shows the ledger entries that result from a single journal posting.

No other journal posting codeunits are required because there is no user interface. Also, you do not have to provide the Seminar Jnl.-Post Batch codeunit, because document posting routines only call the Post Line codeunit.

Exercise 1: Create the Seminar Jnl.-Check Line Codeunit

Exercise Scenario

Create the Seminar Jnl.-Check Line codeunit, which is called from the Seminar Jnl.-Post Line codeunit. This codeunit must perform standard posting checks, such as allowed posting dates, presence of all required fields, and so on.

Note: *If you are unsure how to structure the code in this codeunit, look at the codeunit 211, Res. Jnl-Check Line.*

Task 1: Create the Codeunit

High Level Steps

1. Create the codeunit 123456731, Seminar Jnl.-Check Line, and set the properties to specify the **Seminar Journal Line** as the source table for this codeunit.

Detailed Steps

1. Create the codeunit 123456731, Seminar Jnl.-Check Line, and set the properties to specify the **Seminar Journal Line** as the source table for this codeunit.
   a. In Object Designer, show the codeunits, and then click **New**.
   b. Save the new codeunit as 123456731, Seminar Jnl.-Check Line.
   c. Set the TableNo property for the codeunit to “Seminar Journal Line”.

Task 2: Declare the Variables and Text Constants

High Level Steps

1. Declare the global variables for the **G/L Setup** and **User Setup** tables, and two date variables to keep track of allowed posting period starting and ending dates.
2. Declare the text constants that display errors if entries are posted on closing dates, or outside the allowed posting periods.

Detailed Steps

1. Declare the global variables for the **G/L Setup** and **User Setup** tables, and two date variables to keep track of allowed posting period starting and ending dates.
   a. Declare the following global variables.

Name	DataType	Subtype
GLSetup	Record	General Ledger Setup
UserSetup	Record	User Setup

Name	DataType	Subtype
AllowPostingFrom	Date	
AllowPostingTo	Date	

2. Declare the text constants that display errors if entries are posted on closing dates, or outside the allowed posting periods.
    a. Declare the following global text constants.

Name	ConstValue
Text000	cannot be a closing date.
Text001	is not within your range of allowed posting dates.

Task 3: Create the RunCheck Function

High Level Steps

1. Create the **RunCheck** function that receives a Seminar Journal Line record by reference. Make sure that any code that you add to this function later is enclosed in a WITH block for the record parameter that is passed to the function.
2. From the OnRun trigger, call the **RunCheck** function.

Detailed Steps

1. Create the **RunCheck** function that receives a Seminar Journal Line record by reference. Make sure that any code that you add to this function later is enclosed in a WITH block for the record parameter that is passed to the function.
    a. In the **C/AL Globals** window, create a new function, and name it **RunCheck**.
    b. Define the following parameters for this function.

Var	Name	DataType	Subtype
Yes	SemJnlLine	Record	Seminar Journal Line

    c. In the **RunCheck** function trigger, enter the following code.

```
WITH SemJnlLine DO BEGIN

END;
```

2. From the OnRun trigger, call the **RunCheck** function.
    a. Enter the following code into the OnRun trigger.

```
RunCheck(Rec);
```

Task 4: Add Code to the RunCheck Function

High Level Steps

1. Enter code in the RunCheck function trigger to test whether the **Seminar Journal Line** is empty by using the **EmptyLine** function. If the line is empty, the function exits.
2. Make sure that the **Posting Date**, **Job No.**, and **Seminar No.** fields are not empty.
3. Depending on the value of the **Charge Type** field, make sure that the **Instructor Code**, **Seminar Room Code**, and **Participant Contact No**. fields are not empty.
4. If the line is Chargeable, make sure that the **Bill-to Customer No**. field is not blank.
5. Show an error if the Posting Date is a closing date.
6. Make sure that the **Posting Date** field is between the **Allow Posting From** field and the **Allow Posting To** field values in the **User Setup** table. If these fields are not defined there, then make sure that the **Posting Date** field is between the **Allow Posting From** field and **Allow Posting To** field values in the **G/L Setup** table.
7. Show an error if the **Document Date** field is a closing date, and then save the codeunit.

Detailed Steps

1. Enter code in the RunCheck function trigger to test whether the **Seminar Journal Line** is empty by using the **EmptyLine** function. If the line is empty, the function exits.
   a. In the WITH block of the **RunCheck** function trigger, enter the following code.

```
IF EmptyLine THEN

EXIT;
```

Note: *For all other steps in this task, keep adding the code to the RunCheck function trigger, just before the END of the WITH block.*

2. Make sure that the **Posting Date**, **Job No.**, and **Seminar No.** fields are not empty.
   a. Enter the following code.

```
TESTFIELD("Posting Date");

TESTFIELD("Instructor Resource No.");

TESTFIELD("Seminar No.");
```

3. Depending on the value of the **Charge Type** field, make sure that the **Instructor Code**, **Seminar Room Code**, and **Participant Contact No**. fields are not empty.
   a. Enter the following code.

```
CASE "Charge Type" OF

  "Charge Type"::Instructor:

    TESTFIELD("Instructor Resource No.");

  "Charge Type"::Room:

    TESTFIELD("Room Resource No.");

  "Charge Type"::Participant:

    TESTFIELD("Participant Contact No.");

END;
```

4. If the line is Chargeable, make sure that the **Bill-to Customer No**. field is not blank.
   a. Enter the following code.

```
IF Chargeable THEN

  TESTFIELD("Bill-to Customer No.");
```

5. Show an error if the Posting Date is a closing date.
    a. Enter the following code.

```
IF "Posting Date" = CLOSINGDATE("Posting Date") THEN

  FIELDERROR("Posting Date",Text000);
```

6. Make sure that the **Posting Date** field is between the **Allow Posting From** field and the **Allow Posting To** field values in the **User Setup** table. If these fields are not defined there, then make sure that the **Posting Date** field is between the **Allow Posting From** field and **Allow Posting To** field values in the **G/L Setup** table.
    a. Enter the following code.

```
IF (AllowPostingFrom = 0D) AND (AllowPostingTo = 0D) THEN BEGIN

  IF USERID <> '' THEN

    IF UserSetup.GET(USERID) THEN BEGIN

      AllowPostingFrom := UserSetup."Allow Posting From";

      AllowPostingTo := UserSetup."Allow Posting To";

    END;

  IF (AllowPostingFrom = 0D) AND (AllowPostingTo = 0D) THEN BEGIN

    GLSetup.GET;

    AllowPostingFrom := GLSetup."Allow Posting From";

    AllowPostingTo := GLSetup."Allow Posting To";

  END;

  IF AllowPostingTo = 0D THEN

    AllowPostingTo := 12319999D;

END;

IF ("Posting Date" < AllowPostingFrom) OR ("Posting Date" > AllowPostingTo)
THEN

  FIELDERROR("Posting Date",Text001);
```

Note: *This check is a standard check in all journal posting codeunits. You can take a look at the codeunit 211, Res. Jnl.-Check Line to see how it handles this particular check.*

7. Show an error if the **Document Date** field is a closing date, and then save the codeunit.
   a. Enter the following code.

```
IF ("Document Date" <> 0D) THEN

  IF ("Document Date" = CLOSINGDATE("Document Date")) THEN

    FIELDERROR("Document Date",Text000);
```

   b. Compile, save, and then close the codeunit.

Exercise 2: Create the Seminar Jnl.-Post Line Codeunit

Exercise Scenario

Now you must create the Seminar Jnl.-Post Line codeunit. This executes the core work of the seminar journal posting routine, and creates the **Seminar Ledger Entry** records for the journal posting transaction. This codeunit must handle the following:

- Run the Seminar Jnl.-Check Line codeunit.
- Increase the Entry No. of the ledger entries it creates by one.
- Make sure that one register record is created and maintained throughout the journal posting process to reflect the first and last entry number.
- Insert the ledger entry, and populate it from the fields of the **Seminar Journal Line** table.

Note: *You may want to view the codeunit 212, Res. Jnl.-Post Line to understand the structure and the logic of that codeunit, and then apply the same patterns and concepts in the Seminar Jnl.-Post Line codeunit.*

Task 1: Create the Codeunit

High Level Steps

1. Create the codeunit 123456732, Seminar Jnl.-Post Line, and set the properties to specify the **Seminar Journal Line** as the source table for this codeunit.

Detailed Steps

1. Create the codeunit 123456732, Seminar Jnl.-Post Line, and set the properties to specify the **Seminar Journal Line** as the source table for this codeunit.
   a. In Object Designer, show the codeunits, and then click **New**.
   b. Save the new codeunit as 123456732, Seminar Jnl.-Post Line.
   c. Set the TableNo property for the codeunit to "Seminar Journal Line".

Task 2: Declare the Variables

High Level Steps

1. Declare the global variables for the **Seminar Journal Line**, **Seminar Ledger Entry**, and **Seminar Register** tables. Then create a global variable for the Seminar Jnl.-Check Line codeunit, and an integer variable to keep track of the next available Entry No.

Detailed Steps

1. Declare the global variables for the **Seminar Journal Line**, **Seminar Ledger Entry**, and **Seminar Register** tables. Then create a global variable for the Seminar Jnl.-Check Line codeunit, and an integer variable to keep track of the next available Entry No.
   a. Declare the following global variables.

Name	DataType	Subtype
SeminarJnlLine	Record	Seminar Journal Line
SeminarLedgerEntry	Record	Seminar Ledger Entry
SeminarRegister	Record	Seminar Register
SeminarJnlCheckLine	Codeunit	Seminar Jnl.-Check Line
NextEntryNo	Integer	

Task 3: Create the Functions

High Level Steps

1. Create the **RunWithCheck** and **Code** functions. The **RunWithCheck** function receives a Seminar Journal Line as a parameter by reference.
2. Enter code in the appropriate trigger so that when the program runs codeunit 123456732, Seminar Jnl.-Post Line, it runs the **RunWithCheck** function for the current record.
3. Enter code in the **RunWithCheck** function trigger so that the function copies the *SeminarJnlLine* from the *SeminarJnlLine2* record, runs the **Code** function, and then restores the *SeminarJnlLine2* record back from the *SeminarJnlLine* record.

Detailed Steps

1. Create the **RunWithCheck** and **Code** functions. The **RunWithCheck** function receives a Seminar Journal Line as a parameter by reference.
   a. In the **C/AL Globals** window, create the following functions: **RunWithCheck** and **Code**.
   b. For the **RunWithCheck** function, declare the following parameters.

Var	Name	DataType	Subtype
Yes	SeminarJnlLine2	Record	Seminar Journal Line

2. Enter code in the appropriate trigger so that when the program runs codeunit 123456732, Seminar Jnl.-Post Line, it runs the **RunWithCheck** function for the current record.
   a. In the OnRun trigger, enter the following code.

```
RunWithCheck(Rec);
```

3. Enter code in the **RunWithCheck** function trigger so that the function copies the *SeminarJnlLine* from the *SeminarJnlLine2* record, runs the **Code** function, and then restores the *SeminarJnlLine2* record back from the *SeminarJnlLine* record.
   a. In the **RunWithCheck** function trigger, enter the following code.

```
SeminarJnlLine.COPY(SeminarJnlLine2);

Code;

SeminarJnlLine2 := SeminarJnlLine;
```

Note: *The* SeminarJnlLine *global variable is the main record variable that must be available to all functions in the Seminar Jnl.-Post Line codeunit. By copying it from the by-reference parameter, the whole codeunit has access to the same* ***Seminar Journal Line*** *record.*

When the ***Code*** *function is finished, the by-reference parameter is set to the* SeminarJnlLine *global variable to pass its latest state to the caller. This is a necessary convention because the OnRun trigger is never called directly. It provides backward compatibility only.*

For this codeunit, it is present for convention reasons. The Rec *variable is never initialized, and cannot be used as the global* ***Seminar Journal Line*** *record variable available throughout the codeunit. Therefore, the* SeminarJnlLine *is declared, and the pattern that you see in the* ***RunWithCheck*** *function guarantees the same behavior that you would usually achieve if you called the OnRun trigger directly, and used the* Rec *variable instead.*

Task 4: Add Code to the Code Function

High Level Steps

1. In the **Code** function, enter the WITH code block for the *SeminarJnlLine* record variable.
2. Check whether the *SeminarJnlLine* is empty by using the **EmptyLine** function. If it is empty, the function exits.
3. Runs the **RunCheck** function of the *SeminarJnlCheckLine* codeunit.
4. If the *NextEntryNo* is 0, lock the *SeminarLedgEntry* record, then set the *NextEntryNo* to the **Entry No.** of the last record in the *SeminarLedgEntry* table, if it can be found. Then, increase the *NextEntryNo* by one.
5. If the **Document Date** is empty, the set the **Document Date** to the **Posting Date**.
6. Create or update the **SeminarRegister** record, depending on whether the register record was previously created for this posting. When you create the register record, initialize all fields according to their meaning.
7. Create a new **SeminarLedgerEntry** record, populate the fields from the **SeminarJnlLine** record, set the **Entry No.** field to the *NextEntryNo* variable, insert the new record, and then increment the *NextEntryNo* variable by one. Finally, save the codeunit.

Detailed Steps

1. In the **Code** function, enter the WITH code block for the *SeminarJnlLine* record variable.
   a. In the **Code** function trigger, enter the following code.

```
WITH SeminarJnlLine DO BEGIN



END;
```

2. Check whether the *SeminarJnlLine* is empty by using the **EmptyLine** function. If it is empty, the function exits.
   a. In the WITH block of the **Code** function trigger, enter the following code.

```
IF EmptyLine THEN

  EXIT;
```

Note: *For all successive steps in this task, keep adding the code to the* ***Code*** *function trigger, just before the END of the WITH block.*

3. Runs the **RunCheck** function of the *SeminarJnlCheckLine* codeunit.
   a. Enter the following code.

```
SeminarJnlCheckLine.RunCheck(SeminarJnlLine);
```

4. If the NextEntryNo is 0, lock the SeminarLedgEntry record, then set the NextEntryNo to the **Entry No**. of the last record in the SeminarLedgEntry table, if it can be found. Then, increase the NextEntryNo by one.
   a. Enter the following code.

```
IF NextEntryNo = 0 THEN BEGIN

  SeminarLedgerEntry.LOCKTABLE;

  IF SeminarLedgerEntry.FINDLAST THEN

    NextEntryNo := SeminarLedgerEntry."Entry No.";

  NextEntryNo := NextEntryNo + 1;

END;
```

Note: *If the* NextEntryNo *variable is equal to zero, it means that the* ***Code*** *function was called for the first time during the posting process. In this case, to maintain the transaction integrity, the* ***Seminar Ledger Entry*** *table must be locked.*

Then, the next entry number must be calculated. If there are any other entries in the ***Seminar Ledger Entry*** *table, then the* NextEntryNo *is set to the last* ***Entry No.*** *used, and if there are no other entries, then it remains at zero.*

Finally, the NextEntryNo *is increased by one, to make sure that it either starts at one for the very first ledger entry or at the next available value if there are other ledger entries already in the table.*

5. If the **Document Date** is empty, the set the **Document Date** to the **Posting Date**.
   a. Enter the following code.

```
IF "Document Date" = 0D THEN

  "Document Date" := "Posting Date";
```

6. Create or update the **SeminarRegister** record, depending on whether the register record was previously created for this posting. When you create the register record, initialize all fields according to their meaning.

Note: *If the* ***No.*** *field of the* ***SeminarRegister*** *record is zero, then the register record has not yet been created.*

   a. Enter the following code.

```
IF SeminarRegister."No." = 0 THEN BEGIN

  SeminarRegister.LOCKTABLE;

  IF (NOT SeminarRegister.FINDLAST) OR (SeminarRegister."To Entry No." <> 0)
THEN BEGIN

    SeminarRegister.INIT;

    SeminarRegister."No." := SeminarRegister."No." + 1;

    SeminarRegister."From Entry No." := NextEntryNo;

    SeminarRegister."To Entry No." := NextEntryNo;

    SeminarRegister."Creation Date" := TODAY;
    SeminarRegister."Source Code" := "Source Code";

    SeminarRegister."Journal Batch Name" := "Journal Batch Name";

    SeminarRegister."User ID" := USERID;

    SeminarRegister.INSERT;

  END;

END;

SeminarRegister."To Entry No." := NextEntryNo;

SeminarRegister.MODIFY;
```

***Note:** If the register has not yet been initialized, then the table is first locked to maintain the transaction integrity.*

*If there are no register records in the table at all, or if this is the first time in this transaction that the **Code** function is called (the **To Entry No.** field is zero only for the first call), then a register record is initialized and populated with relevant data. **From Entry No.** is set to the* NextEntryNo.*, which at this point is the first entry for the transaction.*

*Finally, for every call to the **Code** function, the **To Entry No.** field is set to the **NextEntryNo.** field. This increases by one every time that the function is called. This ensures that at the end of the transaction, the **From Entry No.** field of the register record is set to the **Entry No.** field of the first ledger entry. Also, the **To Entry No.** field is set to the **Entry No.** field of the last ledger entry in the transaction.*

7. Create a new **SeminarLedgerEntry** record, populate the fields from the **SeminarJnlLine** record, set the **Entry No.** field to the *NextEntryNo* variable, insert the new record, and then increment the *NextEntryNo* variable by one. Finally, save the codeunit.

   a. Enter the following code.

```
SeminarLedgerEntry.INIT;

SeminarLedgerEntry."Seminar No." := "Seminar No.";

SeminarLedgerEntry."Posting Date" := "Posting Date";

SeminarLedgerEntry."Document Date" := "Document Date";

SeminarLedgerEntry."Entry Type" := "Entry Type";
SeminarLedgerEntry."Document No." := "Document No.";

SeminarLedgerEntry.Description := Description;

SeminarLedgerEntry."Bill-to Customer No." := "Bill-to Customer No.";

SeminarLedgerEntry."Charge Type" := "Charge Type";

SeminarLedgerEntry.Type := Type;

SeminarLedgerEntry.Quantity := Quantity;

SeminarLedgerEntry."Unit Price" := "Unit Price";

SeminarLedgerEntry."Total Price" := "Total Price";

SeminarLedgerEntry."Participant Contact No." := "Participant Contact No.";

SeminarLedgerEntry."Participant Name" := "Participant Name";

SeminarLedgerEntry.Chargeable := Chargeable;

SeminarLedgerEntry."Room Resource No." := "Room Resource No.";

SeminarLedgerEntry."Instructor Resource No." := "Instructor Resource No.";

SeminarLedgerEntry."Starting Date" := "Starting Date";

SeminarLedgerEntry."Seminar Registration No." := "Seminar Registration No.";

SeminarLedgerEntry."Res. Ledger Entry No." := "Res. Ledger Entry No.";

SeminarLedgerEntry."Source Type" := "Source Type";

SeminarLedgerEntry."Source No." := "Source No.";

SeminarLedgerEntry."Journal Batch Name" := "Journal Batch Name";

SeminarLedgerEntry."Source Code" := "Source Code";

SeminarLedgerEntry."Reason Code" := "Reason Code";
SeminarLedgerEntry."No. Series" := "Posting No. Series";

SeminarLedgerEntry."User ID" := USERID;

SeminarLedgerEntry."Entry No." := NextEntryNo;

SeminarLedgerEntry.INSERT;

NextEntryNo := NextEntryNo + 1;
```

b. Compile, save, and then close the codeunit.

Exercise 3: Create the Seminar Ledger Entries Page

Exercise Scenario

Now you must create the page to show the ledger entries. The page must be of list type, and must be noneditable.

Task 1: Create the Page

High Level Steps

1. Create a noneditable list page for the **Seminar Ledger Entry** table by using a wizard and adding the fields to the page.
2. Save the page as **123456721**, **Seminar Ledger Entries**, and close it.

Detailed Steps

1. Create a noneditable list page for the **Seminar Ledger Entry** table by using a wizard and adding the fields to the page.
   a. In Object Designer, show the pages, and then click **New**.
   b. Choose the **Seminar Ledger Entry** table, and then start the List Page Wizard.
   c. Add the following fields to the page:
      - Posting Date
      - Document No.
      - Document Date
      - Entry Type
      - Seminar No.
      - Description
      - Bill-to Customer No.
      - Charge Type
      - Type
      - Quantity
      - Unit Price

        - Total Price
        - Chargeable
        - Participant Contact No.
        - Participant Name
        - Instructor Resource No.
        - Room Resource No.
        - Starting Date
        - Seminar Registration No.
        - Entry No.

    d. Add the **Record Links** and **Notes** system FactBoxes.
    e. Finish the wizard.
    f. Set the Visible property for the Document Date field to *FALSE*.
    g. Set the Editable property for the page to **No**.
    h. Set the Caption property for the page to "Seminar Ledger Entries".

2. Save the page as **123456721**, **Seminar Ledger Entries**, and close it.
    a. Save the page as **123456721, Seminar Ledger Entries**.
    b. Close the **Page Designer** window.

Exercise 4: Create the Seminar Reg.-Show Ledger Codeunit

Exercise Scenario

The last codeunit you must create is the Seminar Reg.-Show Ledger codeunit. The sole purpose of this codeunit is to show the records from the **Seminar Ledger Entry** table that are filtered to only a single transaction. The transaction is defined by the **Seminar Register** record that this function receives through the *Rec* parameter of the OnRun trigger.

Task 1: Create the Codeunit

High Level Steps

1. Create the codeunit **123456734**, **Seminar Reg.-Show Ledger**, and set the properties to specify the **Seminar Register** as the source table for this codeunit.
2. Declare a global variable for the **Seminar Ledger Entry** table.

Detailed Steps

1. Create the codeunit **123456734**, **Seminar Reg.-Show Ledger**, and set the properties to specify the **Seminar Register** as the source table for this codeunit.
    a. In Object Designer, show the codeunits, and then click **New**.

   b. Save the new codeunit as **123456734, Seminar Reg.-Show Ledger**.
   c. Set the TableNo property for the codeunit to “Seminar Register”.

2. Declare a global variable for the **Seminar Ledger Entry** table.
   a. Create the following global variable.

Name	DataType	Subtype
SeminarLedgerEntry	Record	Seminar Ledger Entry

Task 2: Add Code to the OnRun Trigger

High Level Steps

1. Enter code in the appropriate trigger so that when the program runs the codeunit, the codeunit runs the **Seminar Ledger Entries** page. This shows only those entries between the **From Entry No**. field and the **To Entry No.** field on the **Seminar Register**.

Detailed Steps

1. Enter code in the appropriate trigger so that when the program runs the codeunit, the codeunit runs the **Seminar Ledger Entries** page. This shows only those entries between the **From Entry No**. field and the **To Entry No.** field on the **Seminar Register**.
   a. In the OnRun trigger, enter the following code.

```
SeminarLedgerEntry.SETRANGE("Entry No.","From Entry No.","To Entry No.");

PAGE.RUN(PAGE::"Seminar Ledger Entries",SeminarLedgerEntry);
```

   b. Compile, save, and then close the codeunit.

Exercise 5: Create the Seminar Registers Page

Task 1: Create the Page

High Level Steps

1. Create a noneditable list page for the **Seminar Register** table by using a wizard and add the fields to the page.
2. Add an action to the RelatedInformation action container to run the Seminar Reg.-Show Ledger codeunit.
3. Save the page as 123456722, **Seminar Registers**, and close it.

Detailed Steps

1. Create a noneditable list page for the **Seminar Register** table by using a wizard and add the fields to the page.
   a. In Object Designer, show the pages, and then click **New**.
   b. Choose the **Seminar Register** table, and then start the List Page Wizard.
   c. Add the following fields to the page:
      - No.
      - Creation Date
      - User ID
      - Source Code
      - Journal Batch Name
      - From Entry No.
      - To Entry No.

   d. Add the **Record Links** and **Notes** system FactBoxes.
   e. Finish the wizard.
   f. Set the Editable property for the page to **No**.
   g. Set the Caption property for the page to "Seminar Registers".

2. Add an action to the RelatedInformation action container to run the Seminar Reg.-Show Ledger codeunit.
   a. Open the **Page – Action Designer** window.
   b. Define the RelatedInformation action container.
   c. Define the Register action group.
   d. Add the **Seminar Ledger** action, and set RunObject property to run the Seminar Reg.-Show Ledger codeunit.
   e. Assign the WarrantyLedger image to the action and promote it as a large action to the Process category.

3. Save the page as 123456722, **Seminar Registers**, and close it.
   a. Save the page as 123456722, **Seminar Registers**.
   b. Close the **Page Designer** window.

Lab 4.3: Creating the Tables and Pages for Posted Registration Information

Scenario

CRONUS International Ltd. wants to post the Seminar Registration documents after the seminars are completed. You must create the tables and pages that will store and show the posted seminar registration information.

When a user posts a document in Microsoft Dynamics NAV 2013, the structure of the posted information must match the structure of the original information in all relevant aspects. This means that the data model for posted documents must always match the data model for open documents. It is both a convention and a requirement in Microsoft Dynamics NAV 2013 that posted document table fields match the document table fields. If a field is relevant for the posted document, then it must have the same **Field No.** as the same field in the document table. This makes it easier for users to match the posted document information to the information they originally entered into the system. It also simplifies development because you can use the **TRANSFERFIELDS** function to copy the field values between open and posted document tables.

Note: *The* ***TRANSFERFIELDS*** *function copies all fields that have the same* ***Field No.*** *from the source table to the destination table. The fields must have the same data type for the copying to succeed (text and code are convertible, other types are not). There must be room for the actual length of the contents of the field to be copied in the field to which it is to be copied. If any one of these conditions is not fulfilled, a run-time error occurs.*

Therefore, the simplest way to create the posted document tables is by saving the original tables under a different ID and Name. Then make any necessary changes, such as removing unnecessary fields, or appending those fields that are not relevant for the document, but are relevant for the posted document tables.

Exercise 1: Create the Posted Registration Tables

Exercise Scenario

You start by creating the tables for posted registration information. The best approach is to design each document table, and save it under a new ID and name. Then, you must remove all the code from the tables, and make any necessary corrections to table and field properties to meet the requirements and best practices for posted document tables.

Task 1: Create the Posted Seminar Reg. Header Table

High Level Steps

1. Create the **Posted Seminar Reg. Header** table by saving the **Seminar Registration Header** table under ID 123456718.
2. Remove all the code from the table.
3. Delete and rename fields to match the standards for posted document header tables, and add the **User ID** and **Source Code** fields.
4. Add code to the OnLookup trigger of the **User ID** field to run the **LookupUser** function of the User Management codeunit.
5. Correct the calculation formula for the **Comment** field.
6. Set the Caption property for the table to match its Name, and then save and close the table.

Detailed Steps

1. Create the **Posted Seminar Reg. Header** table by saving the **Seminar Registration Header** table under ID 123456718.
   a. Design the table 123456710, **Seminar Registration Header**.
   b. Click **File** > **Save As**. Save the table as 123456718, **Posted Seminar Reg. Header**.

2. Remove all the code from the table.
   a. In the **C/AL Globals** window, delete all variables, text constants, and functions.
   b. In the **C/AL Editor** window, delete all C/AL code. To do this, click the header bar for the Documentation trigger (or any other trigger), then press CTRL+A, then press DEL.
   c. Confirm the deletion by clicking **Yes**.

3. Delete and rename fields to match the standards for posted document header tables, and add the **User ID** and **Source Code** fields.
   a. Delete the **Posting No.** field.
   b. Rename the **Posting No. Series** field to **Registration No. Series**, and modify its Caption accordingly.
   c. Add the following fields.

Field No.	Field Name	Data Type	Length
29	User ID	Code	20
30	Source Code	Code	10

   d. Set the TableRelation property for the **User ID** field to the **User Name** field of the **User** table, and set its ValidateTableRelation and TestTableRelationship properties to **No**.
   e. Set the **TableRelation** property for the **Source Code** field to the **Source Code** table.

4. Add code to the OnLookup trigger of the **User ID** field to run the **LookupUser** function of the User Management codeunit.
   a. In the User ID – OnLookup trigger, view the C/AL Locals.
   b. Define the following local variable.

Name	DataType	Subtype
UserMgt	Codeunit	User Management

   c. Enter the following code.

```
UserMgt.LookupUserID("User ID");
```

5. Correct the calculation formula for the **Comment** field.
   a. In the CalcFormula property for the **Comment** field, correct the table filter so that the **Document Type** field is filtered on Posted Seminar Registration value instead of Seminar Registration value.

Note: *Click the* ***AssistEdit*** *button in the* ***CalcFormula*** *to access the* ***Calculation Formula*** *window, and then click the* ***AssistEdit*** *button in the* ***Table Filter*** *field to access the* ***Table Filter*** *window.*

6. Set the Caption property for the table to match its Name, and then save and close the table.
   a. Set the Caption property for the table to “Posted Seminar Reg. Header”.
   b. Compile, save, and then close the table.

Task 2: Create the Posted Seminar Reg. Line Table

High Level Steps

1. Create the **Posted Seminar Reg. Line** table by saving the **Seminar Registration Line** table under ID 123456719.
2. Correct the table relation for the **Document No.** field.
3. Remove all code from the table.
4. Set the Caption property for the table to match its Name, and then save and close the table.

Detailed Steps

1. Create the **Posted Seminar Reg. Line** table by saving the **Seminar Registration Line** table under ID 123456719.
    a. Design the table 123456711, **Seminar Registration Line**.
    b. Click **File** > **Save As**. Save the table as 123456719, **Posted Seminar Reg. Line**.

2. Correct the table relation for the **Document No.** field.
    a. For the **Document No.** field, set the TableRelation property to the **Posted Seminar Reg. Header** table.

3. Remove all code from the table.
    a. In the **C/AL Globals** window, delete all variables, text constants, and functions.
    b. In the **C/AL Editor** window, delete all C/AL code. To do this, click the header bar for the Documentation trigger (or any other trigger), then press CTRL+A, then press DEL.
    c. Confirm the deletion by clicking **Yes**.

4. Set the Caption property for the table to match its Name, and then save and close the table.
    a. Set the Caption property for the table to "Posted Seminar Reg. Line".
    b. Compile, save, and then close the table.

Task 3: Create the Posted Seminar Charge Table

High Level Steps

1. Create the **Posted Seminar Charge** table by saving the **Seminar Charge** table under ID 123456721.
2. Correct the table relation for the **Document No.** field.
3. Remove all code from the table.
4. Set the Caption property for the table to match its Name, and then save and close the table.

Detailed Steps

1. Create the **Posted Seminar Charge** table by saving the **Seminar Charge** table under ID 123456721.
    a. Design the table 123456712, **Seminar Charge**.
    b. Click **File** > **Save As**. Save the table as 123456721, **Posted Seminar Charge**.

2. Correct the table relation for the **Document No.** field.
    a. For the **Document No.** field, set the TableRelation property to the **Posted Seminar Reg. Header** table.

3. Remove all code from the table.
    a. In the **C/AL Globals** window, delete all variables, text constants, and functions.
    b. In the **C/AL Editor** window, delete all C/AL code. To do this, click the header bar for the Documentation trigger (or any other trigger), then press CTRL+A, then press DEL.
    c. Confirm the deletion by clicking **Yes**.

4. Set the Caption property for the table to match its Name, and then save and close the table.
    a. Set the Caption property for the table to “Posted Seminar Charge”.
    b. Compile, save, and then close the table.

Exercise 2: Import the Posted Registration Pages

Exercise Scenario

After you have created the tables, continue to the most important functionality of the seminar posting feature: the document posting routine. In the meantime, you assign the task to Isaac to develop the following pages for accessing the posted document information: **Posted Seminar Registration**, **Posted Seminar Reg. List**, and **Posted Seminar Charges**.

Task 1: Import the Objects

High Level Steps

1. In Object Designer, import the Mod04\Labfiles\Lab 4.C - Starter - Posted Registration Pages.fob file.
2. Select and compile the imported objects.

Detailed Steps

1. In Object Designer, import the Mod04\Labfiles\Lab 4.C - Starter - Posted Registration Pages.fob file.
    a. In Object Designer, click **File** > **Import**.
    b. In the **Import Objects** dialog box, browse to Mod04\Labfiles\Lab 4.C - Starter - Posted Registration Pages.fob file.
    c. Click **Open**.

2. Select and compile the imported objects.
   a. View all objects in Object Designer.
   b. Filter the view to only show the uncompiled objects.
   c. Compile the objects.
   d. Clear all filters in the Object Designer by pressing SHIFT+CTRL+F7.

Task 2: Review the Objects

High Level Steps

1. Review the imported objects.

Detailed Steps

1. Review the imported objects.
   a. Design the page 123456734, **Posted Seminar Registration**.
   b. Review the page contents in the **Page Preview** window.
   c. Review the page properties.
   d. Close the page 123456734.
   e. Design the page 123456736, **Posted Seminar Reg. List**.
   f. Review the page contents in the **Page Preview** window.
   g. Review the page properties.
   h. Close the page 123456736.
   i. Design the page 123456739, **Posted Seminar Charges**.
   j. Review the page contents in the **Page Preview** window.
   k. Review the page properties.
   l. Close the page 123456739.

Lab 4.4: Modifying Tables, Pages, and Codeunits for Resource Posting

Scenario

When you create new modules, such as Seminar Management, you frequently have to integrate those custom modules with existing features and functionality. Seminars integrate with Resource Management functionality. You use resources to represent instructors and rooms. For auditing and reporting, you want to attach the seminar information to all records in the **Resource Ledger Entry** table. This performs the following goals:

- You have a more robust trail record because you know which resource ledger entries are related to a seminar.
- You can easily and efficiently calculate totals for combinations of instructors, rooms, and seminars.
- You enable seamless user interface flow between Resource Management and Seminar Management functional areas, because all entries are related on the data model level.

You decide to add the following fields to the **Resource Ledger Entry** table.

Field	Remarks
Seminar No.	Lets you keep track of which instructor or room is connected with a seminar.
Seminar Registration No.	Link the posted seminar registration so that users can easily move to all instructor or room resource ledger entries from a posted seminar registration.

Note: *The* ***Seminar No.*** *field seems redundant, because it can be retrieved through the Seminar Registration No. field. However, if you omit the* ***Seminar No.*** *field, you cannot directly filter on resource ledger entries for specific seminars. This reduces the user experience, and adds processing demands when you might have to report or total instructor or room ledger entries by seminar. Therefore, while adding an additional field is not an elegant solution from the data normalization perspective, it is the most efficient solution from the user experience and data processing perspective.*

To make sure that the **Seminar No.** and **Seminar Registration No.** fields are always posted into the **Resource Ledger Entry** table, you must also change the following existing objects.

.Type	ID	Name	Remarks
Table	203	Res. Ledger Entry	
Table	207	Res. Journal Line	To post any new fields to a **Ledger Entry** table, you must first add those fields to the matching **Journal Line** table.
Codeunit	212	Res. Jnl.-Post Line	To move the fields from a **Journal Line** table to the matching **Ledger Entry** table, you must add the appropriate code to the Post Line codeunit for the journal.
Page	202	Resource Ledger Entries	You typically want to show the new fields in the **Ledger Entries** page.

Exercise 1: Modify the Objects

Exercise Scenario

You start by adding and changing the necessary fields to the **Res. Ledger Entry** and **Res. Journal Line** tables, and then modify the **Resource Ledger Entries** page, and the Res. Jnl.-Post Line codeunit.

Task 1: Modify the Res. Ledger Entry Table

High Level Steps

1. Add the **Seminar No**. and **Seminar Registration No**. fields to the table 203, **Res. Ledger Entry**.

Detailed Steps

1. Add the **Seminar No.** and **Seminar Registration No**. fields to the table 203, **Res. Ledger Entry**.
   a. Design the table 203, **Res. Ledger Entry**.
   b. Add the following fields.

No.	Field Name	Type	Length	Remarks
123456700	Seminar No.	Code	20	Set the table relation to the **Seminar** table.
123456701	Seminar Registration No.	Code	20	Set the table relation to the **Posted Seminar Reg. Header** table.

   c. Set the Caption property for both these fields to match their Name.
   d. Compile, save, and then close the table.

Task 2: Modify the Res. Journal Line Table

High Level Steps

1. Add the **Seminar No**. and **Seminar Registration No**. fields to the table 207, **Res. Journal Line**.

Detailed Steps

1. Add the **Seminar No**. and **Seminar Registration No**. fields to the table 207, **Res. Journal Line**.
   a. Design the table 207, **Res. Journal Line**.
   b. Add the following fields.

No.	Field Name	Type	Length	Remarks
123456700	Seminar No.	Code	20	Set the table relation to the **Seminar** table.

No.	Field Name	Type	Length	Remarks
123456701	Seminar Registration No.	Code	20	Set the table relation to the **Posted Seminar Reg. Header** table.

c. Set the Caption property for both these fields to match their **Name**.

d. Compile, save, and then close the table.

Note: *When the fields with same Nos. and Names exist in multiple tables, you can copy them from one table to another. Here, instead of creating fields, you can copy the fields from the* ***Res. Ledger Entry*** *table.*

Task 3: Modify the Resource Ledger Entries Page

High Level Steps

1. Add the **Seminar No**. and **Seminar Registration No**. fields to the page 202, **Resource Ledger Entries**.

Detailed Steps

1. Add the **Seminar No**. and **Seminar Registration No**. fields to the page 202, **Resource Ledger Entries**.
    a. Design the page 202, **Resource Ledger Entries**.
    b. Above the **Job No.** field, insert the **Seminar No.** and **Seminar Registration No.** fields.
    c. Compile, save, and then close the page.

Task 4: Modify the Res. Jnl.-Post Line Codeunit

High Level Steps

1. In codeunit 212, Res Jnl.-Post Line, enter code into the **Code** function trigger so that when the function is populating the **ResLedgEntry** fields, it also assigns the **Seminar No.** and **Seminar Registration No**. fields from the **Res. Journal Line** table.

Detailed Steps

1. In codeunit 212, Res Jnl.-Post Line, enter code into the **Code** function trigger so that when the function is populating the **ResLedgEntry** fields, it also assigns the **Seminar No.** and **Seminar Registration No**. fields from the **Res. Journal Line** table.
    a. Design the codeunit 212, Res. Jnl.-Post Line.

b. In the Code function trigger, below the assignment of the **Qty. per Unit of Measure** field, and above the GetGLSetup line, enter the following code.

```
//CSD1.00>

  ResLedgEntry."Seminar No." := "Seminar No.";

  ResLedgEntry."Seminar Registration No." := "Seminar Registration No.";

//CSD1.00<
```

c. Compile, save, and then close the codeunit.

Lab 4.5: Creating the Codeunits for Document Posting

Scenario

Isaac has started developing the codeunits for seminar registration posting, but the task was too complex for him. He completed the Seminar-Post (Yes/No) codeunit. For the Seminar-Post codeunit he declared variables and functions, and then decided to hand over the task to you. Therefore, you complete the Seminar-Post and Seminar-Post (Yes/No) codeunits that Isaac started developing.

When you have completed the development of these codeunits, you must modify the document pages to let users call the posting routine from the RoleTailored client.

Because you have not yet developed any reports for the Seminar Management functional area, you do not have to provide the Post + Print codeunit.

Exercise 1: Complete the Seminar-Post Codeunit

Exercise Scenario

The Seminar-Post codeunit is the central codeunit of seminar registration posting. It takes the Seminar Registration Header record as a parameter, and processes the information that is contained in it to produce a Posted Seminar Registration document. It must also create the seminar ledger entries for the participants, the instructor, the room, any seminar charges, and the resource ledger entries for the instructor and the room.

Task 1: Import the File

High Level Steps

1. Import the starter objects.

Detailed Steps

1. Import the starter objects.
   a. In Object Designer, click **File** > **Import**.
   b. Locate the Mod04\Labfiles\ Lab 4.E - Starter.fob object file and click **OK**.
   c. Click **Yes** to complete the import.
   d. Close the **Import Objects** window.

Task 2: Complete the CopyCommentLines Function

High Level Steps

1. In the **CopyCommentLines** function trigger, enter the code that finds records in the **Seminar Comment Line** table that matches the specified *FromDocumentType* and *FromNumber*, and for each record inserts a copy of the old record, with the **Document Type** and **No.** set to the ToDocumentType and ToNumber.

Detailed Steps

2. In the **CopyCommentLines** function trigger, enter the code that finds records in the **Seminar Comment Line** table that matches the specified *FromDocumentType* and *FromNumber*, and for each record inserts a copy of the old record, with the **Document Type** and **No.** set to the ToDocumentType and ToNumber.
   a. In the **CopyCommentLines** function trigger, enter the following code.

```
SeminarCommentLine.RESET;

SeminarCommentLine.SETRANGE("Document Type",FromDocumentType);

SeminarCommentLine.SETRANGE("No.",FromNumber);

IF SeminarCommentLine.FINDSET(FALSE,FALSE) THEN BEGIN

  REPEAT

    SeminarCommentLine2 := SeminarCommentLine;

    SeminarCommentLine2."Document Type" := ToDocumentType;

    SeminarCommentLine2."No." := ToNumber;

    SeminarCommentLine2.INSERT;

  UNTIL SeminarCommentLine.NEXT = 0;

END;
```

Task 3: Complete the CopyCharges Function

High Level Steps

1. In the **CopyCharges** function trigger, enter the code that finds all Seminar Charge records that correspond to the specified *FromNumber*. For each record found, the function transfers the values to a new **Posted Seminar Charge** record, by using the **ToNumber** as the **Seminar Registration No**.

Detailed Steps

1. In the **CopyCharges** function trigger, enter the code that finds all Seminar Charge records that correspond to the specified *FromNumber*. For each record found, the function transfers the values to a new **Posted Seminar Charge** record, by using the **ToNumber** as the **Seminar Registration No**.

Note: *Because the* SeminarCharge *and the* PstdSeminarCharge *variables are based on different tables, you cannot assign the record variables directly. All field values must be assigned individually. If the* ***PstdSeminarCharge*** *table field number and types are the same as the* ***SeminarCharge*** *table, you can use the* ***TRANSFERFIELDS*** *function to transfer all the field values at one time.*

a. In the **CopyCharges** function trigger, enter the following code.

```
SeminarCharge.RESET;

SeminarCharge.SETRANGE("Document No.",FromNumber);

IF SeminarCharge.FINDSET(FALSE,FALSE) THEN BEGIN

  REPEAT

    PstdSeminarCharge.TRANSFERFIELDS(SeminarCharge);

    PstdSeminarCharge."Document No." := ToNumber;

    PstdSeminarCharge.INSERT;

  UNTIL SeminarCharge.NEXT = 0;

END;
```

Task 4: Complete the PostResJnlLine Function

High Level Steps

1. In the **PostResJnlLine** function trigger, enter WITH code block for the *SeminarRegHeader* record variable.
2. In the WITH code block, enter the code that does the following:
   - o Makes sure that the **Quantity Per Day** field on the **Resource** record is not empty,
   - o Initializes a **Resource Journal Line** record.
   - o Sets its **Entry Type** to Usage.
   - o Assigns the **Document No.** from the *PstdSeminarRegHeader* record variable.
   - o Assigns the **Resource No.** from the *Resource* record parameter.

3. In the WITH code block, append the code that assigns the following field values from the seminar Registration Header record:
   - o Posting Date
   - o Reason Code
   - o Description
   - o Gen. Prod. Posting Group
   - o Posting No. Series

   Assign these from the fields that have the same name, except for the **Description** field. Assign this from the **Seminar Name** field. Assign the **Source Code** field from the *SourceCode* global variable. Assign the **Resource No.**, **Unit of Measure Code** and **Unit Cost** fields from the *Resource* record parameter. Set the **Qty. per Unit of Measure** field to 1.

4. In the WITH code block, append the code that calculates the **Quantity** field as the product of the **Duration** field from the *SeminarRegHeader* record variable and the **Quantity Per Day** field from the *Resource* record parameter. Then, calculate the **Total Cost** field as the product of the **Unit Cost** and **Quantity** field values. Then, assign values to **Seminar No.** and **Seminar Registration No.** fields. Finally, call the **RunWithCheck** function of the Res. Jnl.-Post Line codeunit.
5. After the WITH block, find the last **Resource Ledger Entry**, and return its **Entry No.** field value as the function return value.

Detailed Steps

1. In the **PostResJnlLine** function trigger, enter WITH code block for the *SeminarRegHeader* record variable.
    b. In the **PostResJnlLine** function trigger, enter the following code.

```
WITH SeminarRegHeader DO BEGIN

END;
```

2. In the WITH code block, enter the code that does the following:
    - Makes sure that the **Quantity Per Day** field on the **Resource** record is not empty,
    - Initializes a **Resource Journal Line** record.
    - Sets its **Entry Type** to Usage.
    - Assigns the **Document No.** from the *PstdSeminarRegHeader* record variable.
    - Assigns the **Resource No.** from the *Resource* record parameter.

    a. In the WITH block, enter the following code.

```
Resource.TESTFIELD("Quantity Per Day");

ResJnlLine.INIT;

ResJnlLine."Entry Type" := ResJnlLine."Entry Type"::Usage;

ResJnlLine."Document No." := PstdSeminarRegHeader."No.";

ResJnlLine."Resource No." := Resource."No.";
```

3. In the WITH code block, append the code that assigns the following field values from the seminar Registration Header record:
    - Posting Date
    - Reason Code
    - Description
    - Gen. Prod. Posting Group
    - Posting No. Series

    Assign these from the fields that have the same name, except for the **Description** field. Assign this from the **Seminar Name** field. Assign the **Source Code** field from the *SourceCode* global variable. Assign the **Resource No.**, **Unit of Measure Code** and **Unit Cost** fields from the *Resource* record parameter. Set the **Qty. per Unit of Measure** field to 1.

a. In the WITH code block, append the following code.

```
ResJnlLine."Posting Date" := "Posting Date";

ResJnlLine."Reason Code" := "Reason Code";

ResJnlLine.Description := "Seminar Name";

ResJnlLine."Gen. Prod. Posting Group" := "Gen. Prod. Posting Group";

ResJnlLine."Posting No. Series" := "Posting No. Series";

ResJnlLine."Source Code" := SourceCode;

ResJnlLine."Resource No." := Resource."No.";

ResJnlLine."Unit of Measure Code" := Resource."Base Unit of Measure";

ResJnlLine."Unit Cost" := Resource."Unit Cost";

ResJnlLine."Qty. per Unit of Measure" := 1;
```

4. In the WITH code block, append the code that calculates the **Quantity** field as the product of the **Duration** field from the *SeminarRegHeader* record variable and the **Quantity Per Day** field from the *Resource* record parameter. Then, calculate the **Total Cost** field as the product of the **Unit Cost** and **Quantity** field values. Then, assign values to **Seminar No.** and **Seminar Registration No.** fields. Finally, call the **RunWithCheck** function of the Res. Jnl.-Post Line codeunit.
   a. In the WITH code block, append the following code.

```
ResJnlLine.Quantity := Duration * Resource."Quantity Per Day";

ResJnlLine."Total Cost" := ResJnlLine."Unit Cost" * ResJnlLine.Quantity;

ResJnlLine."Seminar No." := "Seminar No.";

ResJnlLine."Seminar Registration No." := PstdSeminarRegHeader."No.";
ResJnlPostLine.RunWithCheck(ResJnlLine);
```

5. After the WITH block, find the last **Resource Ledger Entry**, and return its **Entry No.** field value as the function return value.
   a. After the WITH block, enter the following code.

```
ResLedgEntry.FINDLAST;

EXIT(ResLedgEntry."Entry No.");
```

Task 5: Complete the PostSeminarJnlLine Function

High Level Steps

1. In the **PostSeminarJnlLine** function trigger, enter WITH code block for the *SeminarRegHeader* record variable.
2. In the WITH block, enter the code that initializes the *SeminarJnlLine* record variable, and then assigns the following fields from the *SeminarRegHeader* and *PstdSeminarRegHeader* record variables, as appropriate**:**
   - Seminar No.
   - Posting Date
   - Document Date
   - Document No.
   - Charge Type
   - Instructor Resource No.
   - Starting Date
   - Seminar Registration No.
   - Room Resource No.
   - Source Type
   - Source Code
   - Reason Code
   - Posting No.
3. To the WITH code block, append the code that compares the *ChargeType* parameter to all possible option values that it can have.
4. If the *ChargeType* is Instructor, retrieve the appropriate **Resource** record, and then on the *SeminarJnlLine* record variable, assign **Description** from the instructor **Name**, set **Type** to **Resource**, set **Chargeable** to FALSE, and set **Quantity** to the **Duration** field from the **SeminarRegHeader**. Finally, call the *PostResJnlLine*, and assign its return value to the **Res. Ledger Entry No.** field of the *SeminarJnlLine* record variable.
5. If the *ChargeType* is Room, retrieve the appropriate **Resource**, and then on the *SeminarJnlLine* record variable, assign **Description** from the room **Name**, set **Type** to **Resource**, set **Chargeable** to FALSE, and set **Quantity** to the **Duration** field from the **SeminarRegHeader**. Finally, call the **PostResJnlLine**, and assign its return value to the **Res. Ledger Entry No.** field of the *SeminarJnlLine* record variable.

6. If the *ChargeType* is Participant, assign the fields to the *SeminarJnlLine* record variable from the *SeminarRegLine* record variable. Assign the following fields:
   - Bill-to Customer No.
   - Participant Contact No.
   - Participant Name
   - Description
   - Chargeable
   - Unit Price
   - Total Price

   Description is set from **Participant Name**, **Chargeable** is set from **To Invoice**, and **Unit Price** and **Total Price** are set from **Amount**. Set the **Type** to **Resource** and **Quantity** to 1.

7. If *ChargeType* is Charge, then assign the fields to the *SeminarJnlLine* record variable from the *SeminarCharge* record variable. Assign the following fields:
   - Description
   - Bill-to Customer No.
   - Type
   - Quantity
   - Unit Price
   - Total Price
   - Chargeable

   Chargeable is set from **To Invoice**.

8. After the CASE block, post the *SeminarJnlLine* through the Seminar Jnl.-Post Line codeunit.

Detailed Steps

1. In the **PostSeminarJnlLine** function trigger, enter WITH code block for the *SeminarRegHeader* record variable.
   a. In the **PostSeminarJnlLine** function trigger, enter the following code.

```
WITH SeminarRegHeader DO BEGIN

END;
```

2. In the WITH block, enter the code that initializes the *SeminarJnlLine* record variable, and then assigns the following fields from the *SeminarRegHeader* and *PstdSeminarRegHeader* record variables, as appropriate**:**
   - Seminar No.
   - Posting Date
   - Document Date
   - Document No.
   - Charge Type
   - Instructor Resource No.
   - Starting Date
   - Seminar Registration No.
   - Room Resource No.
   - Source Type
   - Source Code
   - Reason Code
   - Posting No.

   a. In the WITH block, enter the following code.

```
SeminarJnlLine.INIT;

SeminarJnlLine."Seminar No." := "Seminar No.";

SeminarJnlLine."Posting Date" := "Posting Date";

SeminarJnlLine."Document Date" := "Document Date";

SeminarJnlLine."Document No." := PstdSeminarRegHeader."No.";

SeminarJnlLine."Charge Type" := ChargeType;

SeminarJnlLine."Instructor Resource No." := "Instructor Resource No.";

SeminarJnlLine."Starting Date" := "Starting Date";

SeminarJnlLine."Seminar Registration No." := PstdSeminarRegHeader."No.";

SeminarJnlLine."Room Resource No." := "Room Resource No.";

SeminarJnlLine."Source Type" := SeminarJnlLine."Source Type"::Seminar;

SeminarJnlLine."Source No." := "Seminar No.";

SeminarJnlLine."Source Code" := SourceCode;
SeminarJnlLine."Reason Code" := "Reason Code";

SeminarJnlLine."Posting No. Series" := "Posting No. Series";
```

3. To the WITH code block, append the code that compares the *ChargeType* parameter to all possible option values that it can have.
   a. In the WITH block, append the following code.

```
CASE ChargeType OF

  ChargeType::Instructor:

    BEGIN

    END;

  ChargeType::Room:

    BEGIN

    END;

  ChargeType::Participant:

    BEGIN

    END;

  ChargeType::Charge:

    BEGIN

    END;

END;
```

4. If the *ChargeType* is Instructor, retrieve the appropriate **Resource** record, and then on the *SeminarJnlLine* record variable, assign **Description** from the instructor **Name**, set **Type** to **Resource**, set **Chargeable** to FALSE, and set **Quantity** to the **Duration** field from the **SeminarRegHeader**. Finally, call the *PostResJnlLine*, and assign its return value to the **Res. Ledger Entry No.** field of the *SeminarJnlLine* record variable.
   a. In the Instructor block, enter the following code.

```
Instructor.GET("Instructor Resource No.");

SeminarJnlLine.Description := Instructor.Name;
SeminarJnlLine.Type := SeminarJnlLine.Type::Resource;

SeminarJnlLine.Chargeable := FALSE;

SeminarJnlLine.Quantity := Duration;

SeminarJnlLine."Res. Ledger Entry No." := PostResJnlLine(Instructor);
```

5. If the *ChargeType* is Room, retrieve the appropriate **Resource**, and then on the *SeminarJnlLine* record variable, assign **Description** from the room **Name**, set **Type** to **Resource**, set **Chargeable** to FALSE, and set **Quantity** to the **Duration** field from the **SeminarRegHeader**. Finally, call the **PostResJnlLine**, and assign its return value to the **Res. Ledger Entry No.** field of the *SeminarJnlLine* record variable.
   a. In the Room block, enter the following code.

```
Room.GET("Room Resource No.");

SeminarJnlLine.Description := Room.Name;

SeminarJnlLine.Type := SeminarJnlLine.Type::Resource;

SeminarJnlLine.Chargeable := FALSE;

SeminarJnlLine.Quantity := Duration;

// Post to resource ledger

SeminarJnlLine."Res. Ledger Entry No." := PostResJnlLine(Room);
```

6. If the *ChargeType* is Participant, assign the fields to the *SeminarJnlLine* record variable from the *SeminarRegLine* record variable. Assign the following fields:
   - Bill-to Customer No.
   - Participant Contact No.
   - Participant Name
   - Description
   - Chargeable
   - Unit Price
   - Total Price

   Description is set from **Participant Name**, **Chargeable** is set from **To Invoice**, and **Unit Price** and **Total Price** are set from **Amount**. Set the **Type** to **Resource** and **Quantity** to 1.

a. In the Participant block, enter the following code.

```
SeminarJnlLine."Bill-to Customer No." := SeminarRegLine."Bill-to Customer No.";

SeminarJnlLine."Participant Contact No." := SeminarRegLine."Participant Contact
No.";

SeminarJnlLine."Participant Name" := SeminarRegLine."Participant Name";

SeminarJnlLine.Description := SeminarRegLine."Participant Name";

SeminarJnlLine.Type := SeminarJnlLine.Type::Resource;

SeminarJnlLine.Chargeable := SeminarRegLine."To Invoice";

SeminarJnlLine.Quantity := 1;

SeminarJnlLine."Unit Price" := SeminarRegLine.Amount;

SeminarJnlLine."Total Price" := SeminarRegLine.Amount;
```

7. If *ChargeType* is Charge, then assign the fields to the *SeminarJnlLine* record variable from the *SeminarCharge* record variable. Assign the following fields:
   - Description
   - Bill-to Customer No.
   - Type
   - Quantity
   - Unit Price
   - Total Price
   - Chargeable

   Chargeable is set from **To Invoice**.

   a. In the Charge block, enter the following code.

```
SeminarJnlLine.Description := SeminarCharge.Description;

SeminarJnlLine."Bill-to Customer No." := SeminarCharge."Bill-to Customer No.";

SeminarJnlLine.Type := SeminarCharge.Type;

SeminarJnlLine.Quantity := SeminarCharge.Quantity;

SeminarJnlLine."Unit Price" := SeminarCharge."Unit Price";

SeminarJnlLine."Total Price" := SeminarCharge."Total Price";
SeminarJnlLine.Chargeable := SeminarCharge."To Invoice";
```

8. After the CASE block, post the *SeminarJnlLine* through the Seminar Jnl.-Post Line codeunit.
    a. After the CASE block, enter the following code.

```
SeminarJnlPostLine.RunWithCheck(SeminarJnlLine);
```

Task 6: Complete the PostCharges Function

High Level Steps

1. In the **PostCharges** function trigger, enter the code that calls the **PostSeminarJnlLine** function for every **Seminar Charge** for the current *SeminarRegHeader*.

Detailed Steps

1. In the **PostCharges** function trigger, enter the code that calls the **PostSeminarJnlLine** function for every **Seminar Charge** for the current *SeminarRegHeader*.
    a. In the PostCharges function trigger, enter the following code.

```
SeminarCharge.RESET;

SeminarCharge.SETRANGE("Document No.",SeminarRegHeader."No.");

IF SeminarCharge.FINDSET(FALSE,FALSE) THEN BEGIN

  REPEAT

    PostSeminarJnlLine(3); // Charge

  UNTIL SeminarCharge.NEXT = 0;

END;
```

Task 7: Add Code to the OnRun Trigger

High Level Steps

1. In the OnRun trigger, enter the code that clears all variables and sets the *SeminarRegHeader* record variable to the current record. Then create a WITH block for the *SeminarRegHeader* variable. After the WITH block, set the current record to the *SeminarRegHeader* record variable.
2. In the WITH block, make sure that the following fields are not empty and that the **Status** field value is Closed:
    o Posting Date
    o Document Date

- Seminar No.
- Duration
- Instructor Resource No.
- Room Resource No.

3. If there are no lines for the current document, throw an error.
4. Open a dialog box to show the posting progress.
5. If the **Posting No.** is blank on the registration header, make sure that the **Posting No. Series** is not blank. Then assign the **Posting No.** to the next number from the posting number series, as indicated on the header. Then, modify the header and perform a commit. Finally, lock the **Seminar Registration Line** table.
6. Assign the *SourceCode* variable from the **Seminar** field of the **Source Code Setup** table.
7. Initialize a new **Posted Seminar Reg. Header** record, and then transfer the fields from the registration header. Assign **No.** and **No. Series** to the **Posting No.** and **Posting No. Series** fields from the registration header. Assign **Source Code** from the *SourceCode* variable, and **User ID** from the **USERID** function. Finally, insert the **Seminar Reg. Header** record.
8. Update the dialog box.
9. Copy the comment lines and charges from the registration header to the posted registration header, by calling the **CopyCommentLines** and **CopyCharges** functions.
10. Set the *LineCount* variable to zero, and prepare the loop for the registration lines of the current registration header.
11. For each registration line, increase the *LineCount* variable by one, update the dialog window, and make sure that **Bill-to Customer No.** and **Participant Contact No.** are not empty. If the line should not be invoiced, reset its **Seminar Price**, **Line Discount %**, **Line Discount Amount** and **Amount** fields to zero. Post the participant line by calling the **PostSeminarJnlLine** function. Finally, initialize and insert a new posted registration line by transferring the fields from the registration line, and assigning the appropriate **Document No.** value.
12. Post the charges by calling the **PostCharges** function. Then post the seminar ledger entry for the instructor and the room by calling the **PostSeminarJnlLine** function.
13. Delete the registration header, lines, comments, and charges.
14. Save the codeunit.

Detailed Steps

1. In the OnRun trigger, enter the code that clears all variables and sets the *SeminarRegHeader* record variable to the current record. Then create a WITH block for the *SeminarRegHeader* variable. After the WITH block, set the current record to the *SeminarRegHeader* record variable.

   a. In the OnRun trigger, enter the following code.

```
CLEARALL;

SeminarRegHeader := Rec;

WITH SeminarRegHeader DO BEGIN



END;

Rec := SeminarRegHeader;
```

2. In the WITH block, make sure that the following fields are not empty and that the **Status** field value is Closed:

   o Posting Date
   o Document Date
   o Seminar No.
   o Duration
   o Instructor Resource No.
   o Room Resource No**.**

Note: *For all remaining steps in this task, always append the code to the end of the WITH block.*

   a. In the WITH block, enter the following code.

```
TESTFIELD("Posting Date");

TESTFIELD("Document Date");

TESTFIELD("Seminar No.");

TESTFIELD(Duration);
TESTFIELD("Instructor Resource No.");

TESTFIELD("Room Resource No.");

TESTFIELD(Status,Status::Closed);
```

3. If there are no lines for the current document, throw an error.

    a. Enter the following code.

```
SeminarRegLine.RESET;

SeminarRegLine.SETRANGE("Document No.","No.");

IF SeminarRegLine.ISEMPTY THEN

  ERROR(Text001);
```

4. Open a dialog box to show the posting progress.

    a. Enter the following code.

```
Window.OPEN(

  '#1#################################\\' +

   Text002);

Window.UPDATE(1,STRSUBSTNO('%1 %2',Text003,"No."));
```

5. If the **Posting No.** is blank on the registration header, make sure that the **Posting No. Series** is not blank. Then assign the **Posting No.** to the next number from the posting number series, as indicated on the header. Then, modify the header and perform a commit. Finally, lock the **Seminar Registration Line** table.

    a. Enter the following code.

```
IF SeminarRegHeader."Posting No." = '' THEN BEGIN

  TESTFIELD("Posting No. Series");

  "Posting No." := NoSeriesMgt.GetNextNo("Posting No. Series","Posting
Date",TRUE);

  MODIFY;

  COMMIT;
```

```
END;

SeminarRegLine.LOCKTABLE;
```

6. Assign the *SourceCode* variable from the **Seminar** field of the **Source Code Setup** table.

    a. Enter the following code.

```
SourceCodeSetup.GET;

SourceCode := SourceCodeSetup.Seminar;
```

7. Initialize a new **Posted Seminar Reg. Header** record, and then transfer the fields from the registration header. Assign **No.** and **No. Series** to the **Posting No.** and **Posting No. Series** fields from the registration header. Assign **Source Code** from the *SourceCode* variable, and **User ID** from the **USERID** function. Finally, insert the **Seminar Reg. Header** record.

    a. Enter the following code.

```
PstdSeminarRegHeader.INIT;

PstdSeminarRegHeader.TRANSFERFIELDS(SeminarRegHeader);

PstdSeminarRegHeader."No." := "Posting No.";

PstdSeminarRegHeader."No. Series" := "Posting No. Series";

PstdSeminarRegHeader."Source Code" := SourceCode;

PstdSeminarRegHeader."User ID" := USERID;

PstdSeminarRegHeader.INSERT;
```

8. Update the dialog box.

    a. Enter the following code.

```
Window.UPDATE(1,STRSUBSTNO(Text004,"No.",

  PstdSeminarRegHeader."No."));
```

9. Copy the comment lines and charges from the registration header to the posted registration header, by calling the **CopyCommentLines** and **CopyCharges** functions.

    a. Enter the following code.

```
CopyCommentLines(
  SeminarCommentLine."Document Type"::"Seminar Registration",
  SeminarCommentLine."Document Type"::"Posted Seminar Registration",
  "No.",PstdSeminarRegHeader."No.");
CopyCharges("No.",PstdSeminarRegHeader."No.");
```

10. Set the *LineCount* variable to zero, and prepare the loop for the registration lines of the current registration header.

    a. Enter the following code.

```
LineCount := 0;
SeminarRegLine.RESET;
SeminarRegLine.SETRANGE("Document No.","No.");
IF SeminarRegLine.FINDSET THEN BEGIN
  REPEAT
  UNTIL SeminarRegLine.NEXT = 0;
END;
```

11. For each registration line, increase the *LineCount* variable by one, update the dialog window, and make sure that **Bill-to Customer No.** and **Participant Contact No.** are not empty. If the line should not be invoiced, reset its **Seminar Price**, **Line Discount %**, **Line Discount Amount** and **Amount** fields to zero. Post the participant line by calling the **PostSeminarJnlLine** function. Finally, initialize and insert a new posted registration line by transferring the fields from the registration line, and assigning the appropriate **Document No.** value.

    a. In the REPEAT block, enter the following code.

```
LineCount := LineCount + 1;
Window.UPDATE(2,LineCount);


SeminarRegLine.TESTFIELD("Bill-to Customer No.");
SeminarRegLine.TESTFIELD("Participant Contact No.");
```

```
IF NOT SeminarRegLine."To Invoice" THEN BEGIN
  SeminarRegLine."Seminar Price" := 0;
  SeminarRegLine."Line Discount %" := 0;
  SeminarRegLine."Line Discount Amount" := 0;
  SeminarRegLine.Amount := 0;
END;

// Post seminar entry
PostSeminarJnlLine(2); // Participant

// Insert posted seminar registration line
PstdSeminarRegLine.INIT;
PstdSeminarRegLine.TRANSFERFIELDS(SeminarRegLine);
PstdSeminarRegLine."Document No." := PstdSeminarRegHeader."No.";
PstdSeminarRegLine.INSERT;
```

12. Post the charges by calling the **PostCharges** function. Then post the seminar ledger entry for the instructor and the room by calling the **PostSeminarJnlLine** function.

    a. After the REPEAT block, enter the following code.

```
// Post charges to seminar ledger
PostCharges;

// Post instructor to seminar ledger
PostSeminarJnlLine(0); // Instructor
```

```
// Post seminar room to seminar ledger

PostSeminarJnlLine(1); // Room
```

13. Delete the registration header, lines, comments, and charges.

    a. Enter the following code.

```
DELETE;

SeminarRegLine.DELETEALL;



SeminarCommentLine.SETRANGE("Document Type",

  SeminarCommentLine."Document Type"::"Seminar Registration");

SeminarCommentLine.SETRANGE("No.","No.");

SeminarCommentLine.DELETEALL;



SeminarCharge.SETRANGE(Description);

SeminarCharge.DELETEALL;
```

14. Save the codeunit.

    a. Compile and save the codeunit, and close the **C/AL Editor** window.

Exercise 2: Enable Posting from the Seminar Registration Pages

Exercise Scenario

After you complete the development of the seminar registration posting routine, you must enable users to start the routine from the relevant pages. In Microsoft Dynamics NAV 2013, users must be able to start posting from the Document and the List pages for documents.

To meet the Microsoft Dynamics NAV 2013 user experience standards, you must add the Post action to the **Seminar Registration** and **Seminar Registration List** pages.

Task 1: Modify the Pages

High Level Steps

1. Add an action to the **Seminar Registration** page that runs the Seminar-Post (Yes/No) codeunit.
2. Add an action to the **Seminar Registration List** page that runs the Seminar-Post (Yes/No) codeunit.

Detailed Steps

1. Add an action to the **Seminar Registration** page that runs the Seminar-Post (Yes/No) codeunit.
   a. Design the page 123456710, **Seminar Registration**.
   b. Add the ActionItems action container.
   c. Add the Posting action group to ActionItems.
   d. Add the **Post** action to Posting group.
   e. Define the following properties:

Property	Value
Caption	P&ost
Image	PostDocument
Promoted	Yes
PromotedCategory	Process
ShortCutKey	F9
RunObject	Codeunit Seminar-Post (Yes/No)

   f. Compile, save, and then close the page.

2. Add an action to the **Seminar Registration List** page that runs the Seminar-Post (Yes/No) codeunit.
   a. Design the page 123456710, **Seminar Registration**.
   b. From the **Page – Action Designer**, select the rows you created in the step 1 and copy them.
   c. Close the **Action Designer**, and the **Page Designer** windows.
   d. Design the page 123456713, **Seminar Registration List**.
   e. In the **Page – Action Designer** page, paste the actions copied in step b.
   f. Close the **Action Designer**.
   g. Compile, save, and then close the page.

Module Review

Module Review and Takeaways

There are two types of posting routines in Microsoft Dynamics NAV 2013: journal posting and document posting routines. These types of posting routines always use the same data model and processing principles, and apply a series of recognizable design patterns. To successfully customize Microsoft Dynamics NAV 2013 and extend it with the new functional areas that support posting routines, you must have a thorough understanding of these standards, and follow them consistently.

A journal in Microsoft Dynamics NAV 2013 consists of at least one of the following:

- The **Journal Line** table if it exists only to support the posting routine.
- The **Journal Batch** and **Journal Template** tables if they enable users to enter information into them from the RoleTailored client.

A journal posting routine in Microsoft Dynamics NAV 2013 consists of at least the Check Line and Post Line codeunits if journals are only posted by the system. You can have several more starter codeunits to handle the user interaction and batch posting, if users manage the journals directly.

Document posting data models consist of the same set of tables as the open (working) documents. At a minimum, this is the Header and the Line table, but may also include any other subsidiary table.

A document posting routine in Microsoft Dynamics NAV 2013 copies the open documents into posted documents, and depends on the **TRANSFERFIELDS** function to simplify the development and maintenance of the posting process. A document posting routine also translates the document information into at least one but frequently many journals, and posts them as an important part of the document posting process. A posted document therefore results not only in the posted document tables, but also in ledger entries.

This module covered the following subjects:

- Posting in Microsoft Dynamics NAV 2013 from journals and from documents
- Tables and codeunits of a standard posting routine
- Key aspects of programming to be aware of to maximize performance

Test Your Knowledge

Test your knowledge with the following questions.

1. Which three tables make up a journal?

2. Which codeunit from the journal posting routine makes sure that data journal lines is complete and valid, before actual posting starts and before any locking occurs?

3. What is the difference between the Post Batch and Batch Post codeunits?

4. Multiline comments can be nested. When using multiline comments you must make sure that each open comments ({) sign is followed by a properly nested close comments (}) sign.

   ( ) True

   ( ) False

5. A table is automatically locked when you start writing data to it.

   ( ) True

   ( ) False

6. Which C/AL function enables you to lock a table immediately and explicitly, even if you make no write access to it?

7. When are the locks released from a locked table?

Test Your Knowledge Solutions

Module Review and Takeaways

1. Which three tables make up a journal?

   MODEL ANSWER:

   Journal Template, Journal Batch, and Journal Line.

2. Which codeunit from the journal posting routine makes sure that data journal lines is complete and valid, before actual posting starts and before any locking occurs?

   MODEL ANSWER:

   Check Line

3. What is the difference between the Post Batch and Batch Post codeunits?

   MODEL ANSWER:

   Post Batch posts lines from a single batch. Batch Post calls Post Batch for each batch selected in the **Journal Batches** page.

4. Multiline comments can be nested. When using multiline comments you must make sure that each open comments ({) sign is followed by a properly nested close comments (}) sign.

   ( ) True

   (√) False

5. A table is automatically locked when you start writing data to it.

   ( ) True

   (√) False

6. Which C/AL function enables you to lock a table immediately and explicitly, even if you make no write access to it?

   MODEL ANSWER:

   LOCKTABLE

7. When are the locks released from a locked table?

   MODEL ANSWER:

   The locks are released at the end of a transaction. A transaction ends automatically when the code execution completes, when you explicitly end it by using the COMMIT function, or when you abort the transaction by using the ERROR function.