|  |  |  |
| --- | --- | --- |
| Solution Overview for | Story: [StoryNumber] – Update Ergon Refining’s product descriptions on all trip history | |
| Author(s) | Kyle Collie |  |
| Date | February 4, 2016 |  |

**Ergon Refining product description**

# Summary

Because we receive Ergon Refining’s waybill information from railroad 417’s instead of railroad 404’s, all of Ergon Refining’s product descriptions in TFM and SRW get a generic descriptor (i.e. Petroleum Lubricating Oil as shown in the below example). This is causing huge problems for Ergon Refining because this descriptor means nothing to them. Instead, they need to know what grade of product is/was in the car. They are taking steps to change this going forward but have requested that we update their current and historical records. This will include completed trips in the Host32 database as well as history in Process Manager.

# Solution

Each car that Ergon ships gets a unique “L #”. Their request is to use this unique L# indicator and a provided spreadsheet, and go through all of Ergon’s current and historical shipments and replace the current generic product descriptor with the more specific Product descriptor that Ergon wants to see (i.e. Hyprene 100; Hyvolt II, etc.)?

According to Michael Sanford, SRWeb uses Host32 data source so the changes would need to take place there.

My research shows that the table and column to update is Host32.TripProduct.Description. The following query will do the trick, assuming that the spreadsheet is the input rather than hard-coding the “L#” and description text.

DECLARE @Desc varchar(200) = 'HYVOLT II C50B'

DECLARE @BOLCustomer varchar(20) = 'L160456'

UPDATE TP

SET TP.[Description] = @Desc

FROM [Host32].[dbo].[TripProducts] TP

  JOIN [Host32].[dbo].[Trips] T

ON T.[CustomerID] = TP.[CustomerID]

  AND T.[TripID] = TP.[TripID]

WHERE T.[CustomerID] = 119

AND T.[BOLCustomer] = @BOLCustomer

In addition, the normal audit columns need to be updated in TripProducts.

Finally, the matching (by TripId) record in Trips needs to have the audit columns updated. This will ensure that Process Manager will be update via normal processing. All the SQL should be contained in one transaction set a,d rolled back if any portion fails.

TripProducts.Description, Trips.UDF2 both need to be updated.

[ErgonTripProduct\_u]

UPDATE TP

SET TP.[Description] = 'SLACK WAX (PAR)',

TP.[UserIDUpdated] = 'QTS-ADMIN',

TP.[WhenUpdated] = GETDATE()

FROM [Host32].[dbo].[TripProducts] TP

JOIN [Host32].[dbo].[Trips] T

ON T.[CustomerID] = TP.[CustomerID]

AND T.[TripID] = TP.[TripID]

WHERE T.[CustomerID] = 119

AND T.[BOLCustomer] = 'L156747'

[ErgonTrip\_u]

UPDATE T

SET T.[UDF2] = 'SLACK WAX (PAR)',

T.[UserIDUpdated] = 'QTS-ADMIN',

T.[WhenUpdated] = GETDATE()

FROM [Host32].[dbo].[Trips] T

WHERE T.[CustomerID] = 119

AND T.[BOLCustomer] = 'L156747'

Example

CREATE PROCEDURE DeleteDepartment

(

@DepartmentID int

)

AS

-- This sproc performs two DELETEs. First it deletes all of the

-- department's associated employees. Next, it deletes the department.

-- STEP 1: Start the transaction

BEGIN TRANSACTION

-- STEP 2 & 3: Issue the DELETE statements, checking @@ERROR after each statement

DELETE FROM Employees

WHERE DepartmentID = @DepartmentID

-- Rollback the transaction if there were any errors

IF @@ERROR <> 0

BEGIN

-- Rollback the transaction

ROLLBACK

-- Raise an error and return

RAISERROR ('Error in deleting employees in DeleteDepartment.', 16, 1)

RETURN

END

DELETE FROM Departments

WHERE DepartmentID = @DepartmentID

-- Rollback the transaction if there were any errors

IF @@ERROR <> 0

BEGIN

-- Rollback the transaction

ROLLBACK

-- Raise an error and return

RAISERROR ('Error in deleting department in DeleteDepartment.', 16, 1)

RETURN

END

-- STEP 4: If we reach this point, the commands completed successfully

-- Commit the transaction....

COMMIT

// This function takes arguments for 2 connection strings and commands to create a transaction

// involving two SQL Servers. It returns a value > 0 if the transaction is committed, 0 if the

// transaction is rolled back. To test this code, you can connect to two different databases

// on the same server by altering the connection string, or to another 3rd party RDBMS by

// altering the code in the connection2 code block.

static public int CreateTransactionScope(

string connectString1, string connectString2,

string commandText1, string commandText2)

{

// Initialize the return value to zero and create a StringWriter to display results.

int returnValue = 0;

System.IO.StringWriter writer = new System.IO.StringWriter();

try

{

// Create the TransactionScope to execute the commands, guaranteeing

// that both commands can commit or roll back as a single unit of work.

using (TransactionScope scope = new TransactionScope())

{

using (SqlConnection connection1 = new SqlConnection(connectString1))

{

// Opening the connection automatically enlists it in the

// TransactionScope as a lightweight transaction.

connection1.Open();

// Create the SqlCommand object and execute the first command.

SqlCommand command1 = new SqlCommand(commandText1, connection1);

returnValue = command1.ExecuteNonQuery();

writer.WriteLine("Rows to be affected by command1: {0}", returnValue);

// If you get here, this means that command1 succeeded. By nesting

// the using block for connection2 inside that of connection1, you

// conserve server and network resources as connection2 is opened

// only when there is a chance that the transaction can commit.

using (SqlConnection connection2 = new SqlConnection(connectString2))

{

// The transaction is escalated to a full distributed

// transaction when connection2 is opened.

connection2.Open();

// Execute the second command in the second database.

returnValue = 0;

SqlCommand command2 = new SqlCommand(commandText2, connection2);

returnValue = command2.ExecuteNonQuery();

writer.WriteLine("Rows to be affected by command2: {0}", returnValue);

}

}

// The Complete method commits the transaction. If an exception has been thrown,

// Complete is not called and the transaction is rolled back.

scope.Complete();

}

}

catch (TransactionAbortedException ex)

{

writer.WriteLine("TransactionAbortedException Message: {0}", ex.Message);

}

catch (ApplicationException ex)

{

writer.WriteLine("ApplicationException Message: {0}", ex.Message);

}

// Display messages.

Console.WriteLine(writer.ToString());

return returnValue;

}

AppDev will create a WinForms application that will allow the specification of i

nput spreadsheet file, reference info such as user name executing the process, the story number documenting the execution, etc. All this information will be written to the standard logging processes. Then the application will read through the spreadsheet and execute a statement similar to the one above to update Host 32 records. Next, the application will execute a similar query to update Process Manager tables.

This WinForms app will be the initial set of applications used as a “tool set” for recurring requests for similar data corrections and for company splits and mergers.

# Estimate

This design is estimated to require 16 hours of effort. Of those 16 hours, 2.5 will be directly associated with the effort for Ergon; the balance will be for QTS foundational tools for future, similar requests.

|  |  |  |  |
| --- | --- | --- | --- |
| Item | Estimate (hours) | Ergon Portion |  |
| Coding | 12 | 1 |  |
| Unit testing | 1 | .5 |  |
| Function & Regression Testing | 2 | 1 |  |
| Administration  (Change requests, etc.) | 1 |  |  |
| Total | 16 | 2.5 |  |