Manuel du L4G SAGE X3

# Introduction et remerciements

Ce manuel a été entièrement écrit par Guillaume CLEMENT en utilisant uniquement mes connaissances et des ressources disponibles en ligne, sans l'aide de robots. La génération du texte a été effectuée intégralement par ChatGPT 4.0 en mars / avril 2024. Le but de ce manuel est principalement de servir à la vectorisation pour alimenter un modèle LLM, dans l’objectif d’être assisté par IA dans les opérations de développement en L4G Sage X3. Ce travail n’est affilié à aucune organisation, peut contenir des erreurs et doit être utilisé avec prudence. Il est totalement libre de tous droits et peut être utilisé comme bon vous semble. Je vous remercie de toute mention de mon travail dans vos réalisations, mais cela n’est nullement une obligation.

https://www.linkedin.com/in/guillaume-clement-erp-cloud/

Les principales sources de données originales sont :

L’aide en ligne Sage X3 <https://online-help.sagex3.com/>

Les dépôts publics sur GitHub, notamment :

<https://github.com/hamydkhadda/L4G>

<https://github.com/pedroarias3x/Sage-X3>

L’aide en ligne Sage X3 accessible via L.V.ExpertiseX3 : <https://lvexpertisex3.com/>

Les sites :

<https://en.sagedev.it/>

<https://www.greytrix.com/>

Merci à toute la communauté Sage X3 sur LinkedIn qui m’a soutenu et encouragé dans ce travail. C’est grâce à vous que j’ai trouvé le courage de réaliser ce manuel.

# Ton rôle

Tu es ChatGPT, le meilleur LLM au monde. Ton entraînement sur le développement informatique, le langage informatique et ta connaissance des bases de données te seront pleinement utiles pour générer du code Sage X3 L4G.

Attention, ton expertise en Oracle et MSSQL peut influencer négativement tes réponses. Tu ne peux pas utiliser les instructions ni les formes de langages propres aux bases de données en Sage X3 L4G.

Tu dois respecter les règles spécifiques à Sage X3 L4G telles que décrites dans ce document. Ton rôle est donc d’assister les développeurs à développer, maintenir et faire évoluer leur code Sage X3 L4G en utilisant ta puissance logique d’enchaînement et d’optimisation du code informatique, tout en t’imposant de respecter les règles de développement en L4G.

# Les variables

## Syntaxe de Déclaration de Variable

La syntaxe générale pour la déclaration de variables en L4G est :

<Étendue> <Type> <Nom de Variable>[(<Dimension>)]

Dans Sage X3, la déclaration d'une variable et son affectation ne peuvent pas être réalisées sur la même ligne directement. Cette contrainte est conçue pour clarifier le code et éviter les erreurs de programmation fréquentes dans des environnements plus permissifs.

### Étendue :

Définit si la variable est locale (Local) ou globale (Global). Les variables locales existent seulement dans le bloc où elles sont définies, tandis que les variables globales restent actives pendant toute la session.

### Type :

Type de la variable, tel que Integer, Char, Date, etc.

Nom de Variable : Le nom attribué à la variable, suivant les conventions de nommage de Sage X3.

### Dimension :

Pour les tableaux, indique le nombre d'éléments ou les limites de l'indexation.

### Exemples de Déclaration :

Integer Local Sans Dimension:

« Local Integer compteur »

Déclare une variable locale de type entier sans dimension spécifique, utilisée généralement pour les compteurs ou autres valeurs numériques simples.

Char Global Avec Dimension:

« Global Char codeProduit[10] »

Déclare une variable globale de type caractère avec une longueur de 10 caractères, utile pour stocker des codes ou des identifiants fixes.

Date Locale:

« Local Date dateInscription »

Déclare une variable locale de type date, utilisée pour enregistrer des dates comme celle d'inscription.

Decimal Local Avec Dimension Multiple:

« Local Decimal prix[10] »

Déclare un tableau de dix éléments de type décimal, qui pourrait être utilisé pour stocker les prix de différents articles.

Tableau Multidimensionnel:

« Local Char noms[5][20] »

Déclare un tableau local de chaînes, où chaque chaîne peut contenir jusqu'à 20 caractères et il y a 5 chaînes dans le tableau. Utile pour stocker des noms ou des petits textes.

Ces exemples illustrent la flexibilité et la précision de la déclaration des variables en L4G, permettant une gestion adaptée des données en fonction des besoins spécifiques de chaque programme ou fonctionnalité dans Sage X3.

## Les variables de base

### Integer :

Description : Représente un nombre entier.

Plage de valeurs : Les entiers peuvent typiquement stocker de grandes valeurs numériques, adaptées aux calculs nécessitant une grande précision sans décimales.

### Shortint :

Description : Similaire à Integer mais utilisé pour des nombres plus petits.

Plage de valeurs : Généralement de -32768 à +32767, ce type est optimal pour économiser la mémoire lorsque les valeurs ne sont pas très grandes.

### libelle (Label) :

Description : Généralement utilisé pour des champs courts de texte comme des noms ou des titres.

Plage de valeurs : Longueur limitée, souvent utilisée pour des textes qui ne dépassent pas 255 caractères.

### Decimal :

Description : Utilisé pour les nombres nécessitant des fractions, comme les montants financiers.

Plage de valeurs : Supporte les nombres avec des décimales, la précision dépendant de la configuration du système.

### Char :

Description : Chaîne de caractères.

Utilisation : Idéal pour stocker du texte de longueur fixe ou variable, comme des adresses ou des descriptions.

### Date :

Description : Utilisé pour stocker des dates.

Format : Les dates sont souvent manipulées à l'aide de fonctions intégrées pour extraire des éléments comme le jour, le mois ou l'année, ou pour effectuer des calculs de date.

### Clbfile :

Description : Type de données pour les grands objets binaires (LOB - Large Object Binary), spécifiquement pour les grands objets de type caractère (CLOB - Character Large Object).

Utilisation : Utilisé pour stocker de grandes quantités de texte, comme des documents ou des données XML.

### Blbfile :

Description : Type de données pour les grands objets binaires (BLOB - Binary Large Object).

Utilisation : Idéal pour stocker des données qui ne sont pas en format texte, comme des images ou des fichiers de données binaires.

# Maîtrise des particularités syntaxiques de Sage X3

La syntaxe de Sage X3, qui est propre au langage L4G utilisé dans cet environnement ERP, présente des spécificités que les développeurs doivent maîtriser pour optimiser la maintenance et le développement d'applications. Voici une exploration approfondie de certaines règles syntaxiques essentielles, avec une attention particulière sur la déclaration et l'affectation des variables.

## Déclaration et Affectation des Variables

Dans Sage X3, la déclaration d'une variable et son affectation ne peuvent pas être réalisées sur la même ligne directement. Cette contrainte est conçue pour clarifier le code et éviter les erreurs de programmation fréquentes dans des environnements plus permissifs.

### Syntaxe Correcte :

Pour déclarer une variable et lui affecter une valeur, il faut utiliser deux instructions distinctes, séparées par le caractère :. Voici comment procéder correctement :

« Local Integer I : I = 1

Local Integer J : J = 2 »

Cette méthode garantit que chaque étape du processus est claire et distincte, facilitant ainsi la lecture et le suivi du code.

## Utilisation de Multiples Instructions sur une Ligne

Bien que chaque instruction doive généralement se trouver sur sa propre ligne, Sage X3 permet de chaîner plusieurs instructions sur une seule ligne à l'aide du caractère :. Cela peut être utile pour regrouper des opérations logiquement connectées sans surcharger visuellement le script.

### Exemple d'Utilisation :

« I = 1 : J = 2 : K = 3 »

Cette ligne affecte des valeurs à trois variables en une seule instruction, montrant comment le chaînage peut être utilisé pour rendre le code plus compact et élégant.

## Commentaires

Les commentaires dans Sage X3 sont introduits par le symbole #, et tout texte suivant ce symbole jusqu'à la fin de la ligne est ignoré par l'interpréteur. Les commentaires sont cruciaux pour expliquer le code, particulièrement lorsqu'il utilise des constructions syntaxiques spécifiques à Sage X3.

Exemple de Commentaire :

« # Initialisation des variables

Local Integer Result : Result = I + J # Calcul de la somme de I et J »

## Extension d'Instructions avec le Caractère "&"

Pour les instructions qui sont trop longues pour tenir sur une seule ligne, Sage X3 utilise le caractère & pour permettre la continuation sur la ligne suivante. C'est crucial pour maintenir la lisibilité dans les scripts plus complexes.

Exemple de Code Multiligne :

« [M:YTX]YFIELD1(nolign-1) = [F:YTM]YFIELD2 + " "

& + " " +func LIB.FUNCTION\_EXAMPLE2(NUMBER1, NUMBER2, NUMBER3, NUMBER4)

& + func LIB.FUNCTION\_EXAMPLE("PARAM1", "PARAM2", "PARAM3", "PARAM4") »

Dans cet exemple, l'instruction est divisée en plusieurs lignes pour améliorer la clarté, chaque nouvelle ligne débutant par &.

### Conclusion

La syntaxe de Sage X3 est conçue pour offrir une structure claire et une forte lisibilité, ce qui est essentiel dans un environnement aussi robuste et complexe que celui d'un ERP. En respectant ces règles syntaxiques, les développeurs peuvent non seulement éviter les erreurs courantes mais aussi faciliter la maintenance et l'extension du code à l'avenir.

## Utilisation des Dates et Heures dans Sage X3

### Fonctions de Base

date$ : Retourne la date courante.

time$ : Retourne l'heure courante sous forme de chaîne.

datetime$ : Combine la date et l'heure actuelles.

### Manipulation de Dates

gdat$(DAY, MONTH, YEAR) : Crée une date à partir du jour, mois, et année spécifiés.

eomonth(date) : Calcule la dernière journée du mois pour une date donnée.

Formatage et Conversion

num$(date$) : Convertit la date courante en chaîne de caractères.

format$("YYYYMMDD", date$) : Formate une date selon le modèle spécifié.

Exemple d'Opérations sur les Dates

« Local date TODAY, LASTDAYOFMONTH

TODAY = date$ # Obtient la date courante

LASTDAYOFMONTH = eomonth(TODAY) #Dernier jour du mois courant »

Infbox num$(LASTDAYOFMONTH) # Affiche la date dans une boîte d'information

### Fonctions Avancées

year(date), month(date), day(date) : Extraient l'année, le mois, et le jour d'une date.

week(date) : Retourne le numéro de la semaine de l'année pour une date donnée.

### Exemples Pratiques

« Local date DATE1

DATE1 = gdat$(1, 1, 2020) # Crée une date pour le 1er janvier 2020

Local integer YEAR, MONTH, DAY

YEAR = year(DATE1)

MONTH = month(DATE1)

DAY = day(DATE1)

Infbox "Année: " + num$(YEAR) + ", Mois: " + num$(MONTH) + ", Jour: " + num$(DAY) »

### Global Variable Formatting

Dans Sage X3, plusieurs variables globales permettent de gérer les formats de date de manière standardisée, ce qui aide à maintenir une cohérence à travers divers utilisateurs et sites.

« # Utilisation de GFMDAT pour formatter une date

func AFNC.FDH(date$, "HHMM") # Formate la date courante avec heures et minutes »

### Conclusion

La gestion des dates et heures dans Sage X3 est robuste, offrant diverses fonctions pour manipuler, formater et calculer des dates de manière efficace. En utilisant les fonctions intégrées, les développeurs peuvent implémenter des fonctionnalités complexes en respectant les normes locales et les exigences spécifiques du système.

## Les variables de Tables

Avant de pouvoir manipuler une table de la base de données il faut que la table soit ouverte.

Pour cela on doit vérifier que la table n’est pas déjà ouverte et l’ouvrir

“If clalev([F:SOP])=0 : Local File SORDERP [F:SOP] : Endif”

Éléments de la syntaxe

If clalev([F:SOP])=0 :

clalev : Cette fonction vérifie si la table ou la classe de fichiers spécifiée est actuellement ouverte dans le contexte du programme en cours d'exécution.

[F:SOP] : Ici, [F:SOP] fait référence à une abréviation de la table que l'on souhaite vérifier. Le préfixe [F:] indique qu'il s'agit d'une classe de fichier (ou table).

=0 : Le résultat de clalev est un entier, où 0 signifie que la table n'est pas ouverte. Si clalev retourne 0, cela signifie que la table avec l'abréviation [F:SOP] n'est pas accessible dans le contexte actuel et peut donc être ouverte.

: Local File SORDERP [F:SOP] :

Local File : C'est l'instruction pour déclarer et ouvrir une table locale. L'utilisation de Local signifie que la table est ouverte uniquement pour le traitement local ou le sous-programme en cours, sans affecter ou interférer avec d'autres parties du programme qui pourraient également utiliser la même table avec des contextes différents.

SORDERP : C'est le nom logique de la table dans le système Sage X3. Ce nom est utilisé pour identifier la table spécifique à ouvrir, ici hypothétiquement représentant une table de lignes de commandes clients.

[F:SOP] : Nouveau, c'est l'abréviation qui sera utilisée pour référencer la table dans le reste du programme. Cette abréviation doit être unique dans le contexte du programme pour éviter les conflits avec d'autres tables ouvertes.

: Endif :

Endif : Termine le bloc conditionnel. Si la condition évaluée par If est fausse (c'est-à-dire que clalev ne retourne pas 0), toutes les instructions entre If et Endif sont ignorées, ce qui signifie que la table ne sera pas ouverte à nouveau si elle est déjà accessible.

Dans la suite du code le fait de déclarer la variable Table me permet de l’utiliser simplement, par son code [F:Abbréviation], dans cet exemple [F:SOP].

### Fermeture d’une table ouverte par Local File :

Par convention on ne ferme que les tables spécifiques à la fin du traitement, les tables qui commencent par le préfixe « Y ». La syntaxe pour fermer un masque est « If clalev([F:ZATV])<>0 : Close File [F:ZATV] : Endif ».

## Les variables de Masques

Dans Sage X3, un masque est un concept qui fait référence à la structure de l'interface utilisateur utilisée pour saisir ou afficher des données. Les masques sont essentiellement des formulaires ou des écrans qui permettent aux utilisateurs d'interagir avec les données de manière structurée. Ils sont utilisés pour définir la présentation et le formatage des données, ainsi que pour contrôler les interactions utilisateur, comme la saisie de données, la modification, et l'affichage.

Compréhension des Masques

Définition :

Un masque en Sage X3 est défini par une série de champs qui peuvent être liés à des colonnes spécifiques dans une base de données ou à des variables du programme. Chaque masque a une mise en page spécifique et peut inclure des éléments tels que des zones de texte, des listes déroulantes, des boutons, etc.

Utilisation :

Les masques facilitent la saisie de données en fournissant un cadre structuré et une validation pour s'assurer que les données saisies sont dans un format acceptable avant d'être traitées ou stockées.

Ils servent aussi à afficher les données de manière organisée pour aider à l'interprétation et à la prise de décision.

Syntaxe d'utilisation des Masques en L4G

If !clalev([M:PTH1]) Local Mask PTH1 [PTH1] : Endif

Explorons les différents éléments de cette instruction :

!clalev([M:PTH1]) :

clalev() est une fonction qui vérifie si le masque ou la classe spécifié est déjà ouvert et accessible dans le contexte actuel du programme.

[M:PTH1] fait référence à un masque identifié par l'abréviation PTH1.

! est l'opérateur de négation, donc !clalev([M:PTH1]) évalue à true si le masque PTH1 n'est pas déjà ouvert.

Local Mask PTH1 [PTH1] :

Local Mask indique la déclaration et l'ouverture d'un masque dans le contexte local. Cela signifie que le masque est utilisé spécifiquement dans la portée du bloc de code actuel ou du sous-programme, sans interférer avec d'autres parties du programme.

PTH1 [PTH1] définit le masque à utiliser, où PTH1 est le nom logique du masque dans Sage X3 et [PTH1] est l'abréviation qui sera utilisée pour référencer les éléments du masque dans le code.

: Endif :

Termine le bloc conditionnel, s'assurant que le masque est ouvert seulement si la condition est vraie.

Conclusion

Les masques dans Sage X3 sont des outils cruciaux pour l'interface utilisateur qui permettent de manipuler et de visualiser les données de manière efficace. Ils assurent que l'interaction de l'utilisateur avec le système est intuitive, sécurisée et conforme aux règles métier définies. La manipulation correcte de l'ouverture et de la fermeture des masques est essentielle pour le développement d'applications robustes et fiables dans l'environnement Sage X3.

### Fermeture d’un masque ouvert par Local Mask :

Par convention on ne ferme que les masques spécifiques à la fin du traitement, les masques qui commencent par le préfixe « Y ». La syntaxe pour fermer un masque est « Close Local Mask [ZCMC2] ».

1. Le formatage du code
   1. Délimiteur de ligne : En L4G, pour passer à une nouvelle ligne au milieu d'une instruction, il faut utiliser le caractère « & » placé en début de la nouvelle ligne, sans espaces avant​​.

Pour avoir plus d’une instruction sur une ligne il faut ajouter le caractère « : » entre les instructions

* 1. Indentation du code : L'indentation n'est pas strictement régulée par des normes syntaxiques en L4G, mais il est couramment accepté d'organiser le code de manière lisible. Par exemple, les blocs conditionnels (If, Else, etc.) et les boucles (For, While, etc.) sont souvent indentés pour améliorer la clarté du code​​.
  2. Règles de syntaxe :

Les noms de variables doivent commencer par une lettre et peuvent inclure jusqu'à 12 caractères alphanumériques​​.

Les variables peuvent être globales ou locales, avec des portées spécifiques décrites dans la documentation​​.

Les instructions telles que Local File sont utilisées pour déclarer des tables, et clalev() pour vérifier si une table est déjà ouverte​​.

Les chaînes de caractères sont concaténées avec le signe +, et les opérations sur les dates et les numériques suivent des formats et fonctions spécifiques, comme date$ pour la date actuelle et num$ pour convertir en chaîne​​.

# Les boucles

## La boucle « For »

En Sage X3 L4G, une boucle "for" est utilisée pour itérer sur une série de valeurs ou pour exécuter un bloc de code un nombre spécifique de fois. Voici une explication détaillée de la syntaxe et du fonctionnement des boucles "for" en L4G, basée sur la documentation disponible.

### Syntaxe de Base

La syntaxe de base d'une boucle "for" en L4G est la suivante :

« For Variable = ValeurDébut To ValeurFin [Step Pas]

... # Instructions à exécuter

Next »

Variable : C'est le compteur de la boucle.

ValeurDébut : La valeur initiale du compteur.

ValeurFin : La valeur finale du compteur.

Pas (optionnel) : Le pas d'incrémentation. Si non spécifié, le compteur est incrémenté de 1.

Exemples

Boucle simple de 1 à 10 :

« For i = 1 To 10

Infbox num$(i)

Next »

Cette boucle affiche les nombres de 1 à 10.

Boucle avec un pas spécifique :

« For i = 1 To 20 Step 2

Infbox num$(i)

Next »

Ici, la boucle affichera les nombres impairs de 1 à 19.

Boucle décrémentale :

« For i = 10 To 1 Step -1

Infbox num$(i)

Next »

Cette boucle compte à rebours de 10 à 1.

### Utilisations Courantes

Les boucles "for" sont souvent utilisées pour manipuler des tableaux, des collections ou pour répéter des opérations un nombre déterminé de fois. Elles sont essentielles pour parcourir des enregistrements de table ou pour des calculs itératifs.

### Considérations

Performance : L'utilisation judicieuse de la boucle "for" est cruciale, car un mauvais design peut entraîner des performances médiocres, surtout avec des bases de données volumineuses.

Contrôle de boucle : La commande Break permet de sortir prématurément d'une boucle "for" si une condition spécifique est remplie, ajoutant ainsi de la flexibilité au contrôle de flux.

Il n’est pas nécessaire de préciser la variable après le Next, par défaut on mettra le « Next » tout seul en commentant simplement la variable de la boucle For pour une meilleure visibilité.

Conclusion

Les boucles "for" en L4G offrent une méthode structurée et répétable pour exécuter des blocs de code multiples fois, ce qui est une fonctionnalité cruciale pour tout développement en Sage X3​​.

## La boucle « While »

Dans le développement Sage X3 L4G, les boucles sont essentielles pour exécuter un bloc de code répétitivement sous certaines conditions. La boucle "while" est l'une des structures de contrôle fondamentales en L4G, utilisée pour répéter une série d'opérations tant qu'une condition spécifiée reste vraie. Cet article explore le fonctionnement de la boucle "while" en L4G, fournissant une compréhension approfondie de son utilisation, des exemples pratiques, et des conseils pour éviter les pièges courants.

### Fonctionnement Basique de la Boucle "While" en L4G

La boucle "while" en L4G permet de répéter des instructions tant que la condition donnée est évaluée à vrai. La structure générale d'une boucle "while" se présente comme suit :

« While [Condition]

... # Instructions à exécuter

Wend »

[Condition] : Expression logique qui est évaluée avant chaque itération de la boucle. Si la condition est vraie, le bloc de code à l'intérieur de la boucle est exécuté. Si la condition est fausse, l'exécution de la boucle se termine et le contrôle est passé à l'instruction suivante après Wend.

### Exemples d'Utilisation

Compteur Simple :

« Local Integer i : i = 1

While i <= 10

Infbox num$(i)

i += 1

Wend »

Cet exemple affiche les nombres de 1 à 10. La boucle continue de s'exécuter tant que i est inférieur ou égal à 10.

### Attente d'une Condition :

« While NOT fichierDisponible

Sleep 1000 # Attendre 1000 millisecondes

Wend »

Ici, la boucle vérifie périodiquement si un fichier est disponible. Elle utilise la fonction Sleep pour réduire la charge du processeur entre les vérifications.

### Bonnes Pratiques

Éviter les Boucles Infinies :

Assurez-vous que la condition de la boucle "while" peut devenir fausse à un moment donné. Une boucle sans une condition de sortie valide peut conduire à une boucle infinie, gelant ainsi l'application.

Optimiser la Condition de la Boucle :

La condition évaluée dans la boucle "while" doit être aussi efficace que possible pour éviter une surcharge inutile. Par exemple, évitez d'appeler des fonctions coûteuses au sein de la condition si elles peuvent être calculées avant la boucle.

Utilisation de Break :

Dans certains cas, il peut être nécessaire de sortir de la boucle avant que la condition de la boucle ne devienne fausse. L'instruction Break permet de sortir immédiatement de la boucle "while".

### Conclusion

La boucle "while" en L4G est un outil puissant pour exécuter un bloc de code de manière répétée sous une condition contrôlée. Elle est particulièrement utile pour les processus d'attente ou les itérations dont le nombre n'est pas connu à l'avance. En suivant les bonnes pratiques et en comprenant clairement comment contrôler les conditions de sortie, les développeurs peuvent utiliser efficacement les boucles "while" pour renforcer la logique de leurs programmes dans Sage X3.

## La boucle « Repeat »

La boucle "repeat" est une structure de contrôle essentielle dans le langage L4G de Sage X3, utilisée pour exécuter un bloc de code jusqu'à ce qu'une condition spécifique soit remplie. Contrairement à la boucle "while", qui teste la condition avant chaque itération, la boucle "repeat" exécute d'abord le bloc de code puis teste la condition à la fin. Cela garantit que le corps de la boucle est exécuté au moins une fois, ce qui est idéal pour des situations où l'itération doit se produire avant que la condition soit évaluée. Cet article explore le fonctionnement de la boucle "repeat" en L4G, illustrant son utilisation avec des exemples et des conseils pratiques.

### Fonctionnement de la Boucle "Repeat" en L4G

La syntaxe de base de la boucle "repeat" en L4G est la suivante :

« Repeat

... # Instructions à exécuter

Until [Condition] »

[Condition] : C'est une expression logique qui est évaluée après l'exécution du bloc de code à l'intérieur de la boucle. Si cette condition est vraie, la boucle s'arrête ; sinon, la boucle se répète.

### Exemples d'Utilisation

Traitement des Utilisateurs jusqu'à Confirmation :

« Local Char choix : choix = "Non"

Repeat

choix = Request("Voulez-vous continuer ? (Oui/Non)")

Until choix = "Oui" »

Cet exemple demande à l'utilisateur s'il souhaite continuer. La boucle se répète jusqu'à ce que l'utilisateur réponde "Oui".

Lecture de Données Jusqu'à Valeur Valide :

« Local Integer nombre

Repeat

nombre = Val(Request("Entrez un nombre positif : "))

Until nombre > 0 »

Ici, la boucle continue de demander à l'utilisateur d'entrer un nombre jusqu'à ce qu'un nombre positif soit fourni.

### Bonnes Pratiques

Assurer une Sortie Possible :

Il est crucial de s'assurer que la condition de sortie puisse être atteinte pour éviter les boucles infinies. Assurez-vous que les actions à l'intérieur du bloc "repeat" modifient les variables impliquées dans la condition pour permettre une évolution vers la sortie.

Minimiser les Opérations Coûteuses :

Comme la boucle "repeat" exécute le bloc de code avant de tester la condition, il est important de minimiser les opérations coûteuses à l'intérieur de la boucle pour ne pas pénaliser les performances, surtout si la condition de sortie prend du temps à être satisfaite.

Clarté de la Condition de Sortie :

La condition de sortie doit être claire et directement liée aux actions effectuées dans le bloc de la boucle. Cela améliore la lisibilité et la maintenance du code.

### Conclusion

La boucle "repeat" en L4G est un outil puissant pour les scénarios où il est nécessaire d'exécuter un bloc de code au moins une fois avant de tester une condition de sortie. Elle est particulièrement utile dans les cas de validation d'entrée ou lorsque certaines actions doivent précéder la décision de continuer ou d'arrêter une répétition. En utilisant judicieusement cette structure, les développeurs peuvent construire des logiques de programme robustes et efficaces dans Sage X3.

## L’instruction « Break »

L'instruction Break dans le langage L4G de Sage X3 est un outil essentiel pour gérer le flux de contrôle dans les boucles, notamment dans les boucles For, While, et Repeat. Cette instruction permet d'interrompre immédiatement une boucle et de passer à l'instruction qui suit immédiatement la boucle. C'est particulièrement utile pour sortir prématurément d'une boucle lorsqu'une condition spécifique est remplie, sans attendre que la condition de la boucle elle-même soit fausse.

### Fonctionnement de l'Instruction Break

Dans une boucle, lorsque le flux d'exécution atteint l'instruction Break, la boucle est immédiatement terminée, et le contrôle du programme saute à l'instruction qui suit la boucle. Break est souvent utilisé en combinaison avec une instruction conditionnelle If pour tester une condition spéciale qui nécessite une sortie anticipée de la boucle.

### Exemples d'Utilisation

« For i = 1 To 100

If i = 50 Then

Break

Endif

Infbox num$(i) # Affiche les nombres de 1 à 49

Next »

Dans cet exemple, la boucle For est interrompue quand i atteint 50. L'exécution continue avec l'instruction suivante après la boucle.

Utilisation dans une Boucle While pour Éviter une Opération :

« Local Integer compteur : compteur = 1

While compteur <= 100

If compteur = 25 Then

Break # Sort de la boucle si compteur est 25

Endif

compteur += 1

Wend »

Ici, la boucle While est interrompue lorsque compteur atteint 25. Cela empêche l'exécution des opérations qui pourraient suivre dans la boucle.

### Bonnes Pratiques

Éviter l'Usage Excessif :

Bien que Break soit utile, son usage excessif peut rendre le flux de contrôle difficile à suivre, ce qui complique la maintenance du code. Il est préférable de l'utiliser judicieusement.

### Clarté des Conditions de Sortie :

Assurez-vous que les conditions sous lesquelles Break est utilisé sont clairement commentées ou évidentes pour éviter la confusion sur la raison pour laquelle la boucle est interrompue.

### Alternatives à Break :

Dans certains cas, restructurer la condition de la boucle ou utiliser une structure de contrôle différente peut être plus clair que d'utiliser un Break.

### Conclusion

L'instruction Break dans L4G de Sage X3 offre un contrôle précis sur le flux d'exécution dans les boucles, permettant une sortie anticipée sous certaines conditions. Elle doit être utilisée avec prudence pour maintenir la lisibilité et la maintenabilité du code. En suivant les meilleures pratiques et en utilisant Break de manière réfléchie, vous pouvez efficacement gérer des situations complexes dans vos scripts L4G.

### « break x » ; sortir de plusieurs niveaux d’imbrication

L'instruction Break "x" dans le langage L4G de Sage X3 est une extension de l'instruction Break classique. Elle est utilisée pour sortir de plusieurs niveaux de boucles imbriquées en une seule opération. C'est une fonctionnalité puissante qui permet de gérer des situations complexes où des boucles multiples sont enchevêtrées et où une condition dans une boucle intérieure pourrait nécessiter une sortie immédiate de plusieurs niveaux de boucles.

Fonctionnement de Break "x"

Lorsque vous utilisez Break suivi d'un identifiant entre guillemets, comme Break "x", vous spécifiez un point de sortie ciblé pour les boucles imbriquées. L'identifiant "x" doit correspondre à une étiquette que vous avez préalablement définie au niveau d'une boucle englobante. Lorsque Break "x" est exécuté, le contrôle du programme saute immédiatement à l'instruction qui suit la boucle marquée par cette étiquette.

Définir des Étiquettes pour les Boucles

Pour utiliser Break "x", vous devez d'abord marquer vos boucles avec des étiquettes. Voici un exemple illustrant comment définir ces étiquettes :

« Label "outerLoop"

For i = 1 To 10

Label "innerLoop"

For j = 1 To 10

If i \* j > 50 Then

Break "outerLoop" # Sort de toutes les boucles jusqu'à l'étiquette "outerLoop"

Endif

Next »

Next »

Dans cet exemple, si le produit de i et j dépasse 50, Break "outerLoop" interrompt non seulement la boucle interne (For j) mais aussi la boucle externe (For i), grâce à l'utilisation de l'étiquette "outerLoop".

### Conseils d'Utilisation

Clarté et Maintenance : Utilisez des noms d'étiquettes qui sont descriptifs et cohérents pour faciliter la lecture et la maintenance du code. Les étiquettes doivent clairement indiquer la portée ou le but de la boucle qu'elles marquent.

Documentation : Commentez abondamment l'utilisation de Break "x" et les étiquettes associées pour expliquer pourquoi une sortie prématurée est nécessaire. Cela aide les autres développeurs à comprendre le flux de contrôle sans devoir déchiffrer la logique complexe des conditions.

Utilisation Prudente : Comme Break "x" peut rendre le flux de contrôle du programme plus difficile à suivre, utilisez-le judicieusement. Assurez-vous qu'il n'y a pas d'alternative plus simple ou plus claire avant de recourir à des sorties multiples.

### Conclusion

Break "x" est un outil avancé dans L4G pour gérer les sorties de boucles multiples efficacement, ce qui est particulièrement utile dans des scénarios de traitement imbriqués complexes. Toutefois, son utilisation nécessite une attention particulière à la clarté du code et à la documentation pour assurer que le code reste lisible et maintenable. En respectant ces pratiques, vous pouvez tirer parti de cette fonctionnalité puissante sans compromettre la qualité de votre code.

# Gestion des Erreurs et Avertissements dans Sage X3 : Utilisation de Onerrgo et Resume

La gestion des erreurs est un aspect crucial du développement de logiciels, particulièrement dans les environnements ERP comme Sage X3, où les processus d'affaires sont complexes et intégrés. Sage X3 offre des mécanismes sophistiqués pour la gestion des erreurs, notamment à travers les instructions Onerrgo et Resume. Cette section explique en détail ces mécanismes, leur utilisation, et donne des exemples pour illustrer leur application pratique.

## Principe de Fonctionnement de Onerrgo et Resume

Onerrgo est une instruction utilisée pour rediriger le flux d’exécution du programme vers une étiquette spécifique en cas d’erreur. Cela permet de gérer les erreurs de manière contrôlée en exécutant une section de code dédiée au traitement de ces erreurs.

### Syntaxe de Base :

« Onerrgo <Etiquette> [From <traitement>] »

Resume est utilisée pour revenir au traitement initial après l’instruction qui a causé l'erreur. Cela permet de reprendre l'exécution du programme après que l'erreur ait été gérée.

### Exemples d'Utilisation

Exemple 1 : Gestion Simple d'Erreur

Supposons que vous ayez besoin d'ouvrir un fichier pour la lecture. Vous pouvez utiliser Onerrgo pour gérer les cas où le fichier ne peut pas être ouvert (par exemple, s'il n'existe pas).

« Onerrgo ERREUR

Openi filpath("TXT", "ZETEST", "txt")

...

Onerrgo

End

$ERREUR

Infbox "Erreur n°" - errn - errmes$(errn) - errm

Infbox "Erreur dans traitement" - errp - "en ligne" - errl

Resume »

Dans cet exemple :

Onerrgo ERREUR : Redirige vers l’étiquette ERREUR en cas d'échec de l'ouverture du fichier.

Infbox : Affiche des informations sur l'erreur, utilisant des fonctions pour récupérer des détails comme le numéro de l'erreur (errn), le message d'erreur (errmes$), etc.

Resume : Reviendra à la ligne qui a tenté d'ouvrir le fichier une fois l'erreur gérée.

Exemple 2 : Suppression de la Gestion d'Erreur

« Onerrgo # Suppression de toute gestion d'erreur précédemment définie. »

Utiliser Onerrgo sans étiquette supprime toute redirection d'erreur établie précédemment, rétablissant le comportement par défaut qui est l'arrêt du traitement en cas d'erreur.

### Considérations Importantes

Terminaison du Traitement d'Erreur : Si le traitement de gestion d'erreur se termine par End, le traitement du programme principal est interrompu de façon définitive.

Gestion des Transactions : S'il y a une transaction en cours quand une erreur est détectée, et que le traitement de gestion d'erreur se termine par End, la transaction est automatiquement annulée (Rollback).

Réinitialisation de la Gestion d'Erreur : Utiliser Onerrgo sans étiquette pour supprimer la redirection vers une étiquette en cas d'erreur, rétablissant le flux normal en cas d'erreurs non gérées.

### Conclusion

L'utilisation de Onerrgo et Resume dans Sage X3 permet une gestion des erreurs robuste et flexible, essentielle pour maintenir l'intégrité des processus d'affaires complexes. En intégrant ces commandes dans vos programmes, vous pouvez assurer que les erreurs sont non seulement capturées, mais aussi gérées de manière appropriée, minimisant ainsi les perturbations et maximisant la fiabilité du système.

# Manipulation des données, « Where », « Filter », »Order by »

Dans le développement Sage X3 en utilisant le langage L4G, les clauses Where, Filter, et Order By sont essentielles pour la manipulation et la consultation des données stockées dans des tables. Ces clauses permettent de préciser les conditions de sélection, de filtrage et de tri des enregistrements lors de requêtes sur les bases de données. La compréhension de leur utilisation est cruciale pour optimiser les performances des applications et maintenir la clarté du code.

## Utilisation de la Clause Where

La clause Where est utilisée pour spécifier les conditions que les enregistrements doivent satisfaire pour être sélectionnés ou affectés par une opération. Elle est typiquement utilisée dans les instructions For, Select, et Filter pour limiter les données manipulées à celles qui répondent à des critères spécifiques.

Syntaxe :

« For [F:Table] Where [Condition]  »

## Utilisation de la Clause Filter

La clause Filter est utilisée pour appliquer un filtre dynamique sur une table, souvent en combinaison avec la fonction evalue() qui permet d'évaluer une expression ou une variable contenant des critères de filtrage sous forme de texte.

« Filter [F:Table] Where Expression]  »

Ou pour des cas plus complexes : « Filter [F:Table] Where evalue([Expression]) »

## Utilisation de la Clause Order By

La clause Order By est utilisée pour trier les enregistrements récupérés d'une table selon un ou plusieurs champs, spécifiés dans l'ordre ascendant ou descendant.

« For [F:Table] Order By [Champ] Asc|Desc »

## Cumuler des Clauses avec AND ou &

Pour combiner plusieurs conditions dans une même clause Where ou Filter, on peut utiliser AND ou &. Ces opérateurs permettent de lier des conditions logiques pour qu'un enregistrement doive les satisfaire toutes pour être sélectionné.

## Exemples Pratiques

«

CRITERE = "PLI='TFRSC'"

CRITERE+= " & PLICRI1='"+[M:PIH0]BPR+"'"

CRITERE+= " & PLICRI2='"+[M:PIH3]ITMREF(NUML)+"'"

CRITERE+= " & PLICRI3='"+vireblc(num$(WTYPPAL),2)+"'"

CRITERE+= " & PLISTRDAT<=["+format$("D:DD[/]MM[/]YYYY",[M:PIH0]ACCDAT)+"]"

CRITERE+= " & PLIENDDAT>=["+format$("D:DD[/]MM[/]YYYY",[M:PIH0]ACCDAT)+"]"

Filter [F:PPL] Where evalue(CRITERE) »

Dans cet exemple, une variable CRITERE est construite dynamiquement pour contenir une série de conditions filtrantes, et est ensuite évaluée dans une clause Filter. Cela permet de filtrer dynamiquement les données de la table [F:PPL] selon plusieurs critères.

Exemple 2

«

Filter [F:ZMH] Where [F:ZMH]ZBPSNUM=[M:POH0]BPSNUM and [F:ZMH]ZBPAADD=[M:POH0]ZCODADR and [F:ZMH]ZAXEMET=[M:POH1]CCE(1) and [F:ZMH]ZTYPMOD=3) »

Cette expression filtre la table [F:ZMH] pour des enregistrements qui satisfont toutes les conditions données, utilisant des valeurs directement prises de masques de l'écran.

Exemple 3

«

For [F:ZTCFG]ZTCFG0 Where [F:ZTCFG]POHNUM=[M:POH0]POHNUM »

Ici, une boucle For est utilisée pour itérer sur des enregistrements de la table [F:ZTCFG] qui correspondent à une condition spécifique.

Exemple 4

«

Filter[F:ZPTR] Where [F:ZPTR]ZATNUM=WZATNUM

WNBENR = rowcount([F:ZPTR])

Filter[F:ZPTR]

If(WNBENR>0)Then

RESULT = 2

Else

RESULT = 1

Endif

End RESULT »

Dans cet exemple, la table [F:ZPTR] est d'abord filtrée par une condition, puis un test est réalisé pour vérifier si des enregistrements existent après le filtrage. Selon le résultat, une valeur est attribuée à la variable RESULT.

Conclusion

Les clauses Where, Filter, et Order By en L4G permettent une gestion fine des données, essentielle pour les performances et l'exactitude des applications Sage X3. Leur utilisation appropriée assure que seules les données pertinentes sont traitées, améliorant ainsi l'efficacité et la clarté du code.

# Opération CRUDE en base de données

Dans le développement Sage X3 en utilisant le langage L4G (4th Generation Language), vous avez accès à diverses instructions pour interagir avec la base de données, notamment pour lire, créer, modifier et supprimer des enregistrements. Les commandes de base incluent Read, Write, Rewrite, et Delete. Chacune de ces opérations joue un rôle crucial dans la gestion des données au sein de l'ERP Sage X3. Voici un guide détaillé sur leur utilisation avec des exemples concrets.

## Lire des Données (Read)

Read est utilisée pour récupérer des enregistrements d'une base de données basés sur une clé d'index spécifiée. La syntaxe standard est :

« Read [F:Table]Index = Valeur »

[F:Table] : Représente l'abréviation de la table.

Index : La clé d'index spécifiée pour la recherche.

Valeur : La valeur de la clé utilisée pour localiser l'enregistrement.

Importance des Indices dans Read

Utiliser le bon index est crucial pour les performances de la requête. Un index mal choisi peut entraîner des recherches lentes, surtout avec des bases de données volumineuses. Chaque utilisation de Read doit être consciente de l'index employé.

### Pièges et Précautions

Performance :

S'assurer que l'index utilisé est optimisé pour les requêtes effectuées.

### Gestion des erreurs :

Vérifier fstat après chaque Read pour s'assurer que l'enregistrement est trouvé, sinon gérer l'absence de données de manière appropriée.

Initialisation des données :

Réinitialiser les variables de table avant un Read pour éviter de traiter des données erronées en cas de non-trouvaille.

### Exemples Pratiques avec Indices

Exemple 1: Recherche de Partenaire Commercial

« Local File BPARTNER [BPR]

Read [BPR]BPR0=12345 # Utilisation de l'index BPR0 pour la recherche

If (fstat<>0) Then

Infbox "Partenaire non trouvé"

Else

Infbox "Nom du partenaire : " + [BPR]NAME

EndIf »

Dans cet exemple, l'index BPR0 est utilisé pour rechercher un partenaire commercial spécifique. L'efficacité de Read dépend de l'optimisation de cet index.

### Exemple 2: Mise à jour d'adresse client

« Local File CUSTOMER [CST]

Read [CST]CST0="CUST001" # Index CST0 utilisé pour identifier le client

If (fstat=0) Then

[CST]ADDRESS = "Nouvelle adresse"

Rewrite [CST]

If (fstat<>0) Then

Infbox "Erreur de mise à jour"

EndIf

Else

Infbox "Client non trouvé"

EndIf »

### Exemple 3 : Cas concret

« Local File PRODUCT [PRD]

Read [PRD]PRD1="SKU123" # Index PRD1 pour la référence produit

If (fstat=0) Then

Infbox "Détails du produit : " + [PRD]DESCRIPTION

Else

Infbox "Produit non trouvé"

EndIf »

Cet exemple, issu de la base de connaissances, montre comment utiliser l'index PRD1 pour charger les détails d'un produit spécifique.

### Conclusion

L'instruction Read est un outil puissant pour la manipulation de données dans Sage X3, mais son efficacité dépend largement de la compréhension et de l'utilisation appropriée des indices. En suivant les meilleures pratiques et en gérant correctement les erreurs, les développeurs peuvent garantir des opérations de base de données rapides et fiables.

### L'utilisation des index multicolumnes dans le contexte Sage X3 L4G

Dans les scripts L4G de Sage X3, l'utilisation d'index multicolumnes est une technique avancée pour optimiser les performances des requêtes de base de données, particulièrement lors de l'accès à des enregistrements spécifiques basés sur plusieurs critères. Cet article détaille l'utilisation des index multicolumnes à travers des exemples concrets issus du fichier "various scripts".

### Avantages des Index Multicolonnes

Les index multicolumnes améliorent significativement l'efficacité des requêtes qui nécessitent de filtrer ou de trier selon plusieurs colonnes. Par exemple, une requête qui doit récupérer des données basées sur à la fois le code client et le code produit sera plus rapide si un index combinant ces deux colonnes existe.

### Exemple Pratique avec Index Multicolonne

Dans le script suivant, extrait du fichier "various scripts", un index multicologne est utilisé pour interroger efficacement les données :

« # Extrait de la configuration de lecture avec index multicologne

Read [F:ZPM]ZPM0=[M:POH0]BPSNUM;[M:POH0]ZCODADR;[M:POH0]POHFCY »

Cet exemple montre l'utilisation d'un index multicologne ZPM0 pour la table ZPM, où les colonnes BPSNUM, ZCODADR, et POHFCY sont combinées pour une recherche rapide. Cela permet de filtrer efficacement les enregistrements selon ces trois critères simultanément, réduisant ainsi le temps de réponse pour les opérations de base de données.

### Utilisation d'une Variable pour la Jointure

Utiliser des variables dans les requêtes permet une flexibilité accrue, notamment en rendant les scripts plus dynamiques et adaptables aux conditions de runtime. Voici un exemple issu du même fichier :

« # Utilisation de variable pour filtrer avec index multicologne

Local Char POHFCY[10]

POHFCY = "001"

Read [F:ZPM]ZPM0=[M:POH0]BPSNUM;[M:POH0]ZCODADR;POHFCY »

Dans cet exemple, la variable POHFCY est utilisée pour spécifier dynamiquement une partie de l'index multicologne dans une opération Read. Cette méthode est particulièrement utile dans les scénarios où la valeur de la colonne peut varier en fonction des conditions d'exécution ou des entrées utilisateur.

### Conclusion

Les index multicolumnes sont essentiels pour optimiser les performances des applications Sage X3 en réduisant le temps nécessaire pour accéder aux données dans des tables volumineuses. En combinant plusieurs colonnes dans un seul index et en utilisant des variables pour les jointures, les développeurs peuvent réaliser des scripts plus performants et adaptatifs.

## Créer des Données (Write)

La commande Write est utilisée pour créer un nouvel enregistrement dans une table.

### Syntaxe de Base :

« [F:Table]Champ = Valeur

Write [F:Table] »

Exemple :

« [F:BPC]BPCNUM = "CUST124"

[F:BPC]NAME = "New Customer"

Write [F:BPC] »

Ce code crée un nouveau partenaire commercial avec un identifiant "CUST124" et un nom "New Customer".

## Modifier des Données (Rewrite)

La commande Rewrite est utilisée pour modifier un enregistrement existant. Vous devez d'abord lire l'enregistrement, modifier les champs nécessaires, puis appliquer Rewrite.

### Syntaxe de Base :

« Read [F:Table]Key = Valeur

[F:Table]Champ = NouvelleValeur

Rewrite [F:Table] »

Exemple :

« Read [F:BPC]BPC0 = "CUST123"

[F:BPC]NAME = "Updated Customer"

Rewrite [F:BPC] »

Ce code lit l'enregistrement du partenaire commercial "CUST123", modifie le nom en "Updated Customer", puis met à jour l'enregistrement dans la base de données.

## Effacer des Données (Delete)

La commande Delete est utilisée pour supprimer un enregistrement existant de la base de données. Comme pour Rewrite, il est souvent nécessaire de lire l'enregistrement avant de le supprimer.

### Syntaxe de Base :

« Read [F:Table]Key = Valeur

Delete [F:Table] »

Exemple :

« Read [F:BPC]BPC0 = "CUST124"

Delete [F:BPC] »

Ce code lit l'enregistrement du partenaire commercial "CUST124" et le supprime ensuite de la base de données.

Conseils Généraux

Optimisation : Utilisez des clés et des index adéquats pour optimiser les performances des opérations de lecture et d'écriture.

En utilisant ces commandes de manière appropriée, vous pouvez efficacement gérer les données dans votre application Sage X3, garantissant ainsi que les informations sont correctement créées, lues, mises à jour et supprimées selon les besoins de l'entreprise.

## Fonctionnement de Delete avec Where

L'instruction Delete avec Where permet de supprimer tous les enregistrements qui correspondent aux conditions spécifiées, sans avoir besoin de les lire explicitement avant la suppression. Cela rend l'opération de suppression plus rapide et moins coûteuse en termes de performance, surtout lorsque vous travaillez avec de grandes quantités de données.

### Syntaxe de Base :

« Delete [F:Table] Where [Condition] »

Exemple Expliqué

Voici un exemple typique de l'utilisation de Delete avec Where :

« Delete [F:ZZS] Where [F:ZZS]ZSEQ=[F:ZZSA]ZSEQ

If(fstat<>0)Then

WERRTRAIT = 1

INFO = "[3.17] - ERREUR Suppression"

If WFICTRA : Call ECR\_TRACE(INFO,1) From GESECRAN : Endif

Endif »

Delete [F:ZZS] Where [F:ZZS]ZSEQ=[F:ZZSA]ZSEQ : Cette ligne supprime tous les enregistrements de la table [F:ZZS] où le champ ZSEQ correspond à la valeur de ZSEQ dans une autre table ou variable [F:ZZSA]. Cela peut être utilisé pour synchroniser ou nettoyer les données entre deux tables reliées par une clé commune.

Gestion des erreurs :

If(fstat<>0)Then : Après la tentative de suppression, la variable système [S]fstat est vérifiée. Une valeur différente de zéro indique une erreur lors de l'opération de suppression.

WERRTRAIT = 1 : Une variable est définie pour indiquer qu'une erreur de traitement a eu lieu.

INFO = "[3.17] - ERREUR Suppression" : Un message d'erreur est préparé pour être loggé ou affiché.

If WFICTRA : Call ECR\_TRACE(INFO,1) From GESECRAN : Endif : Si le traçage est activé (WFICTRA), le message d'erreur est enregistré dans le système de trace. ECR\_TRACE est une fonction qui écrit des informations de diagnostic dans un fichier de trace ou sur un écran de contrôle.

Recommandations

Utilisation de Delete avec Where : C'est recommandé pour les opérations de suppression en masse pour améliorer les performances et réduire la charge de travail sur le serveur.

Gestion des erreurs : Toujours vérifier le résultat de [S]fstat après une opération de suppression pour gérer les erreurs efficacement. Cela aide à maintenir l'intégrité des données et à informer les utilisateurs ou les systèmes de log des problèmes potentiels.

Documentation et logging : Assurez-vous que toutes les opérations de suppression importantes sont documentées et que les erreurs sont loggées pour faciliter le débogage et la maintenance du système.

En suivant ces pratiques, vous pouvez assurer que les suppressions de données dans votre application Sage X3 sont réalisées de manière efficace, sécurisée et conforme aux besoins opérationnels de l'entreprise.

## Fonctionnement de l'Instruction Link

L'instruction Link dans Sage X3 est une fonctionnalité puissante permettant de créer des jointures entre deux tables ou plus. Cette commande est cruciale pour optimiser les requêtes de données complexes, en réduisant le besoin de boucles imbriquées et en simplifiant les accès aux données. Cependant, sa syntaxe peut prêter à confusion et requiert une compréhension claire pour éviter les erreurs courantes.

### Syntaxe de Base du Link

La syntaxe générale pour créer une jointure avec Link est la suivante :

« Link [F:TablePrincipale] With [F:TableLiee]KEY0~=FIELD1TableLiee;FIELD2TableLiee As [Abreviation]

& Where [Conditions]

& Order By [Critères de Tri] »

Table Principale et Tables Liées : Avant de définir un Link, il est impératif que toutes les tables impliquées soient ouvertes. Sage X3 ne permet pas d'ouvrir les tables dans le cadre de l'instruction Link ; elles doivent être ouvertes au préalable dans le script.

### Type de Jointure :

Utiliser ~= pour créer une jointure stricte (INNER JOIN).

Utiliser = pour une jointure extérieure gauche (LEFT JOIN).

Abreviation de Lien : L'abréviation (alias) du lien doit être unique et sert uniquement à la lecture. Il n'est pas possible de modifier les données à travers cette abréviation. Pour les mises à jour ou suppressions, il faut utiliser les abréviations des tables originales.

### Limitations Importantes :

Un Link peut inclure jusqu'à 8 tables liées.

L'abréviation de lien ne contient que les champs de la table principale.

Il est impossible d'écrire ou de supprimer des données via l'abréviation de lien.

Les clauses Where utilisées dans le Link doivent être placées avant la boucle For et non à l'intérieur.

La jointure ne peut pas être utilisée avec une classe [G] associée.

Sage X3 DB2 a une limitation sur la lecture de fichiers qui dépassent 255 colonnes; une attention particulière est nécessaire lors de la création de Link pour éviter cette limitation.

### Exemples Pratiques

### Exemple de Jointure Interne :

Supposons que nous devons lier les tables des commandes (SORDER) et des lignes de commande (SORDERP) pour filtrer certaines commandes spécifiques :

« Link [F:SORDER] With [F:SORDERP]SORDNUM~=SORDNUM As [JOINTURE]

& Where [F:SORDERP]ITMREF = "TOTO"

& Order By [F:SORDER]SORDNUM Asc »

### Exemple de Jointure Gauche :

Pour afficher toutes les commandes, y compris celles sans lignes correspondantes dans SORDERP :

« Link [F:SORDER] With [F:SORDERP]SORDNUM=SORDNUM As [JOINTURE]

& Where [F:SORDER]STATUS = "Active"

& Order By [F:SORDER]SORDNUM Asc »

### Conclusion

L'utilisation correcte de l'instruction Link dans Sage X3 augmente l'efficacité et la performance des scripts en réduisant le besoin de multiples itérations sur les ensembles de données. Toutefois, les développeurs doivent être vigilants quant à la syntaxe et les limitations, notamment l'impossibilité d'utiliser des clauses « Where » dans les boucles « For » liées à un lien et la gestion prudente des jointures pour éviter les erreurs de performance. L'apprentissage et l'application correcte de ces principes sont essentiels pour exploiter pleinement les capacités de Sage X3.

L'Utilisation Impérative de l'Instruction Read avec un Index dans Sage X3 L4G

Dans le langage L4G de Sage X3, l'instruction Read est un élément essentiel pour accéder efficacement aux données stockées dans les bases de données de l'ERP. L'utilisation d'un index avec l'instruction Read n'est pas seulement une pratique recommandée mais une nécessité impérative pour optimiser les performances et assurer la précision des opérations de base de données. Cet article détaille le fonctionnement de Read, son interaction avec les index, et pourquoi son utilisation avec un index est obligatoire dans Sage X3.

# Comprendre l'Instruction Read

L'instruction Read en L4G permet de récupérer un enregistrement spécifique d'une table en utilisant une clé d'index prédéfinie. La syntaxe générale de cette commande est :

« Read [F:NomTable]Index = Valeur1; Valeur2; ... »

[F:NomTable] : C'est l'abréviation de la table dans laquelle la recherche est effectuée.

Index : La clé d'index spécifiée pour la recherche.

Valeur1; Valeur2; ... : Les valeurs correspondant aux différents champs de l'index, utilisées pour localiser l'enregistrement désiré.

## Pourquoi l'Index est-il Impératif ?

L'utilisation d'un index avec Read est impérative pour plusieurs raisons cruciales :

Accès Rapide aux Données : Les index permettent un accès rapide et efficace aux données. Ils fonctionnent comme une table des matières pour la base de données, permettant au système de trouver rapidement l'enregistrement exact sans avoir à parcourir toute la table. Cela est particulièrement vital dans les environnements de production où la rapidité d'accès aux données peut affecter directement la performance opérationnelle.

Optimisation des Performances : Les bases de données dans Sage X3 peuvent devenir très volumineuses avec des millions d'enregistrements. Sans index, chaque requête Read nécessiterait un balayage complet de la table, ce qui serait extrêmement inefficace et consommateur de temps. L'utilisation d'index réduit considérablement le temps nécessaire pour accéder aux données, ce qui améliore directement la performance globale de l'application.

Précision et Fiabilité : L'utilisation d'index assure que les requêtes sont non seulement rapides mais également précises. En spécifiant les valeurs d'index, Read extrait exactement l'enregistrement requis selon les critères définis, ce qui minimise les erreurs potentielles de récupération des données erronées.

## Exemple Pratique

Considérons une table CLIENT avec un index composite CLI0 composé des champs ClientID et ClientRegion. Pour récupérer un enregistrement spécifique, la syntaxe de Read serait :

« Read [F:CLIENT]CLI0 = 123; "Nord" »

Dans cet exemple, 123 est la valeur du ClientID et "Nord" est la valeur de ClientRegion. Cette commande récupère l'enregistrement où ces deux conditions sont simultanément satisfaites, utilisant l'index CLI0 pour une récupération rapide et précise.

## Conclusion

L'instruction Read est une composante fondamentale de la programmation en L4G pour Sage X3, permettant aux développeurs d'accéder de manière efficace et précise aux données. L'utilisation obligatoire d'un index avec Read n'est pas un simple choix de conception mais une exigence pour garantir des performances optimales, une précision accrue et une efficacité maximale dans la manipulation des données. En respectant cette contrainte, les développeurs peuvent assurer que leurs applications Sage X3 sont robustes, réactives et fiables.

# Gestion des Statuts d'Opérations sur les Tables avec fstat dans Sage X3

Dans Sage X3, la variable système fstat est essentielle pour surveiller le succès ou l'échec des opérations sur les tables. Elle est mise à jour après chaque opération de base de données telle que Read, Write, Rewrite, et Delete. Cette variable indique le statut de l'opération dernière exécutée, fournissant un feedback crucial pour la gestion des erreurs et le contrôle de flux dans les scripts L4G.

## Utilisation de fstat :

Read: Après une opération de lecture, fstat permet de vérifier si un enregistrement a été trouvé ou non. Si fstat vaut 0, l'enregistrement a été trouvé et lu avec succès. Si fstat est différent de zéro, cela peut indiquer que l'enregistrement n'existe pas ou qu'une autre erreur s'est produite.

Write: Lors de l'écriture d'un enregistrement, fstat vérifie si l'écriture a été réalisée avec succès. Un fstat de zéro indique un succès, tandis qu'une valeur non nulle peut indiquer une erreur comme un conflit de verrouillage ou une violation de contrainte.

Rewrite: Similaire à Write, cette opération est utilisée pour mettre à jour un enregistrement existant. fstat servira à déterminer si la mise à jour a été effectuée sans erreurs.

Delete: Pour les opérations de suppression, fstat indique si l'enregistrement ciblé a été supprimé correctement. Une valeur non nulle pourrait signifier que l'enregistrement n'existait pas ou était verrouillé.

## Importance de fstat :

La surveillance de fstat est cruciale pour la robustesse et la fiabilité des applications Sage X3. Elle permet aux développeurs de gérer correctement les scénarios d'erreur et de s'assurer que les opérations de base de données se déroulent comme prévu. Cela aide à éviter des erreurs silencieuses où un problème lors d'une opération pourrait passer inaperçu, conduisant à des données incorrectes ou à des états d'application incohérents.

En résumé, fstat est un outil indispensable dans le développement L4G pour Sage X3, garantissant que toutes les interactions avec la base de données sont surveillées et contrôlées avec précision. Cela renforce l'intégrité des données et la gestion des erreurs dans les applications Sage X3.

# Manipulation des variables

## Fonctionnement du Trans-Classe

Dans le développement sur Sage X3, l'utilisation de la fonctionnalité Trans-Classe est essentielle pour simplifier et optimiser la synchronisation des données entre les tables et les masques de l'application. Cette technique permet une copie efficace des données d'une table à une autre, ou entre une table et un masque, en réduisant la redondance du code et en accélérant les processus de développement. Cet article explore les particularités du Trans-Classe dans Sage X3, illustrant son utilisation pratique et ses avantages.

Le Trans-Classe est un mécanisme dans Sage X3 qui permet de transférer automatiquement les valeurs de champs d'une table à ceux d'une autre, ou entre une table et un masque, en utilisant une correspondance de noms de champs. Cette méthode est particulièrement utile pour les opérations qui nécessitent la réplication de données entre différentes structures de données dans l'ERP.

Exemple d'Utilisation du Trans-Classe

Supposons que vous devez charger des données du premier enregistrement de la table BPARTNER dans un masque correspondant :

« Local File BPARTNER [BPR] # Ouverture de la table BPARTNER

Read [F:BPR]BPR0 first # Lecture du premier enregistrement de la table

If [S]fstat = 0 # Si la lecture est bonne

# Assigner les valeurs de la table aux champs du masque : 1ère solution

[M:BPR1] BPRSHO = [F:BPR]BPRSHO

[M:BPR1] BPRNAM(0) = [F:BPR]BPRNAM(0)

[M:BPR1] BPRNAM(1) = [F:BPR]BPRNAM(1)

[M:BPR1]CRY = [F:BPR]CRY

# Assigner les valeurs de la table aux champs du masque : 2nde solution

[M:BPR1] = [F:BPR]  »

Dans cet exemple, la première solution montre une affectation manuelle champ par champ, tandis que la deuxième solution utilise le Trans-Classe pour affecter automatiquement tous les champs correspondants entre la table BPR et le masque BPR1.

### Avantages du Trans-Classe

Réduction de Redondance : Le Trans-Classe réduit la nécessité de coder explicitement chaque affectation de champ, minimisant ainsi la redondance et les erreurs potentielles.

Maintenance Simplifiée : En modifiant la structure d'une table ou d'un masque, il n'est pas nécessaire de réviser les programmes qui utilisent le Trans-Classe, tant que les noms de champs restent cohérents.

Automatisation : Les champs dimensionnés et ceux soumis à des codes d'activité sont automatiquement alimentés sans tests préalables, facilitant la gestion des données complexes.

### Considérations Complémentaires

Bien que le Trans-Classe offre des avantages significatifs, il est important de gérer les cas où les noms de champs diffèrent entre les tables ou les masques, ou lorsque des champs spécifiques n'existent pas dans la source ou la destination. Dans ces situations, une alimentation manuelle champ par champ peut être nécessaire pour compléter ou ajuster les données transférées.

### Conclusion

Le Trans-Classe dans Sage X3 est un outil puissant pour les développeurs, leur permettant de gérer efficacement les données entre différentes structures de l'application. En utilisant cette fonctionnalité, les développeurs peuvent non seulement accélérer le développement et la maintenance des applications mais aussi garantir une intégrité et une cohérence accrues des données au sein de l'ERP.

## Affectation Dynamique de Variables avec Assign

Dans le développement logiciel, notamment dans des environnements ERP tels que Sage X3, la flexibilité dans la gestion des variables peut être cruciale pour traiter efficacement les données dynamiques. Sage X3, avec son langage L4G, offre des fonctionnalités avancées pour manipuler des variables de manière dynamique à travers les fonctions Assign et Evalue. Cet article explore ces fonctionnalités, expliquant comment elles permettent de gérer des noms de variables et des valeurs de manière flexible et puissante.

La fonction Assign en L4G est utilisée pour attribuer des valeurs à des variables dont les noms sont construits dynamiquement pendant l'exécution. Cette fonctionnalité est particulièrement utile dans les scénarios où le nom de la variable dépend d'autres variables ou conditions qui ne sont déterminées qu'au moment de l'exécution.

### Comment ça marche ?

Assign permet de construire le nom d'une variable sous forme de chaîne de caractères, puis d'assigner une valeur à cette variable. Le format général de la commande est :

« Assign "NomVariable" With Valeur »

### Exemple Pratique

Supposons que vous avez besoin de charger des données dans une série de champs qui sont numérotés séquentiellement (par exemple, REP0, REP1, ..., REP10). Au lieu de répéter le code d'affectation pour chaque champ, vous pouvez utiliser une boucle et Assign :

« For I= 0 to 10

Assign "[M:BPC4]REP" + num$(I) + "(nolign-1)" With [F:BPD]REP(I)

Next »

Dans cet exemple, [M:BPC4]REP représente une base de nom de variable à laquelle le numéro de champ et l'indice de ligne sont ajoutés dynamiquement. [F:BPD]REP(I) est la valeur assignée à chaque champ, où I varie de 0 à 10.

### Lecture Dynamique de Variables avec Evalue

La fonction Evalue complète Assign en permettant la lecture de variables dont les noms sont également construits dynamiquement. Cette fonction évalue une expression qui contient un nom de variable et retourne sa valeur.

### Comment ça marche ?

Evalue est utilisée pour accéder à la valeur d'une variable dont le nom est formulé comme une chaîne de caractères :

« Variable = Evalue("NomVariable") »

### Exemple Pratique

Considérons que vous avez besoin de lire des valeurs à partir de champs dynamiquement nommés comme décrit précédemment :

« For I= 0 to 1

[F:BPD]REP(I) = Evalue("[M:BPC4]REP" + num$(I) + "(nolign-1)")

Next »

Ici, Evalue permet d'accéder aux valeurs des champs [M:BPC4]REP0(nolign-1) et [M:BPC4]REP1(nolign-1), en utilisant un indice qui change dans la boucle.

### Conclusion

Les fonctions Assign et Evalue offrent une puissante capacité de manipulation dynamique des variables dans Sage X3, simplifiant le code et augmentant sa flexibilité. Ces méthodes sont essentielles pour les développeurs qui travaillent avec des données dont la structure peut varier pendant l'exécution, permettant des scripts plus dynamiques et adaptatifs. En maîtrisant ces outils, les développeurs peuvent améliorer considérablement la maintenabilité et la clarté de leurs solutions ERP.

## Gestion des CLOBs dans Sage X3

Dans l'environnement de développement Sage X3, la manipulation des types de données complexes comme les BLOBs (Binary Large Objects) et les CLOBs (Character Large Objects) est une capacité essentielle, particulièrement dans les contextes où la gestion de grands volumes de données textuelles ou binaires est nécessaire. Cet article explore la manière dont Sage X3 gère ces types de données à travers des exemples de déclaration et d'utilisation des variables BLOB et CLOB.

Un CLOB, ou Character Large Object, est utilisé pour stocker de grandes quantités de texte, ce qui est idéal pour gérer des documents ou des données qui excèdent les limites des types de données standards.

Déclarations de Variables CLOB

La déclaration de variables CLOB dans Sage X3 peut être effectuée de plusieurs manières, en fonction des besoins spécifiques en termes de capacité ou de structure des données :

Déclaration Standard :

« Local Clbfile WWCLOB »

Cette commande crée une variable CLOB avec une longueur par défaut définie par le système.

### Déclaration avec Longueur Spécifique :

« Local Clbfile WWCLOB(3) »

Ici, le CLOB est configuré pour contenir jusqu'à 4094 caractères, ce qui est utile pour des cas où la taille des données est connue et limitée.

Déclaration de Tableaux de CLOBs :

« Local Clbfile WWCLOB(3)(2)

Local Clbfile WWCLOB(3)(0..1)  »

Ces déclarations préparent des tableaux de CLOBs, permettant de gérer plusieurs objets de texte de grande taille de manière structurée.

Manipulation de Variables CLOB

La manipulation des CLOBs permet de gérer efficacement des données textuelles complexes :

### Assignation et Copie :

« [M:XXX]CLOB = [F:XXX]CLOB

[M:XXX] = [F:XXX] »

Ces lignes copient des données CLOB entre différentes variables, facilitant le transfert et la duplication de grandes quantités de texte.

### Concaténation avec des Chaînes de Caractères :

« Local Char TEXTE(250)(1..100)

For I=1 to 100

[M:XXX]CLOB += TEXTE(I)

Next »

Cette boucle illustre comment concaténer des chaînes à un CLOB, ce qui est pratique pour construire des documents ou des logs progressivement.

Échange de Données entre CLOBs et Chaînes :

« Setlob TEXTE With [M:XXX]CLOB

Setlob [M:XXX]CLOB With TEXTE(1..100) »

Ces opérations montrent comment les données peuvent être transférées entre des CLOBs et des variables de type chaîne, permettant des manipulations flexibles des données textuelles.

### Avantages des CLOBs en Version 5

Avec la version 5 de Sage X3, les CLOBs sont traités de manière encore plus intégrée, comme des chaînes de caractères standards. Cette amélioration permet d'utiliser toutes les instructions normales de manipulation de chaînes sur des CLOBs, simplifiant le code et rendant les scripts plus lisibles.

Conclusion

La gestion des BLOBs et CLOBs dans Sage X3 fournit une flexibilité essentielle pour les applications qui nécessitent une manipulation intensive de données textuelles ou binaires. Que ce soit pour stocker des documents, gérer de grands ensembles de données ou réaliser des opérations complexes sur des textes volumineux, les fonctionnalités L4G de Sage X3 rendent ces tâches efficaces et gérables.

Dans l'environnement Sage X3, la manipulation des blocs tableaux sur les écrans est une tâche commune pour les développeurs qui cherchent à gérer et présenter des données de manière structurée et interactive. La gestion des variables dans un contexte de bloc tableau est particulièrement importante, car elle permet de dynamiser les interactions avec les données de l'ERP. Cet article explore comment les variables et la méthode Trans-Classe sont utilisées pour gérer efficacement les données dans les blocs tableaux d’un écran Sage X3.

## Manipulation des Données dans les Blocs Tableaux

Exemple de Configuration et de Chargement

Considérons un scénario où nous devons charger et afficher des informations sur les clients livrés dans un bloc tableau d’un écran :

« # Initialisation

raz [M:BPC4]

[S]nolign=0

# Chargement des données

For [BPD]BPD0 Where [F]BPCNUM=[M:BPC0]BPCNUM

[S]nolign += 1

[M:BPC4] = [F:BPD]

[M:BPC4]BPDNAM0(nolign-1) = [F:BPD]BPDNAM(0)

[M:BPC4]BPDNAM1(nolign-1) = [F:BPD]BPDNAM(1)

# Gestion des champs optionnels

If dim([M:BPC4]REP0)>0 [M:BPC4]REP0(nolign-1) = [F:BPD]REP(0) endif

If dim([M:BPC4]REP1)>0 [M:BPC4]REP1(nolign-1) = [F:BPD]REP(1) endif

Next

# Mise à jour de la variable de bas de tableau

[M:BPC4]NBDLV = nolign »

Dans cet exemple, nous utilisons une boucle For pour parcourir les enregistrements pertinents dans la table BPD. Chaque ligne correspondante est chargée dans le masque BPC4 avec des ajustements de l'index de ligne réalisés via la variable [S]nolign. Cette approche permet une alimentation ligne par ligne dans le tableau.

### Trans-Classe et Alimentation Champ par Champ

Le Trans-Classe est utilisé ici pour simplifier la copie des données entre la table source et le masque de l'écran. Cependant, dans des cas où les noms des champs dans la source et la destination ne correspondent pas exactement, ou lorsque des champs spécifiques requièrent un traitement conditionnel, une alimentation champ par champ devient nécessaire. Cette méthode est également utilisée pour gérer les champs dimensionnés et conditionnels :

« # Alimentation conditionnelle basée sur la présence et la dimension des champs

If dim([M:BPC4]REP0)>0 [M:BPC4]REP0(nolign-1) = [F:BPD]REP(0) endif

If dim([M:BPC4]REP1)>0 [M:BPC4]REP1(nolign-1) = [F:BPD]REP(1) endif »

### Avantages et Recommandations

L'utilisation du Trans-Classe et de l'alimentation champ par champ dans les blocs tableaux offre plusieurs avantages :

Efficacité : Le Trans-Classe permet une copie rapide et efficace des données sans avoir besoin de coder explicitement chaque transfert de champ.

Flexibilité : L'alimentation champ par champ offre la flexibilité nécessaire pour gérer des exceptions, comme des champs absents ou des conditions spéciales sur certains champs.

Maintenabilité : Simplifier le code en utilisant le Trans-Classe réduit la complexité et améliore la maintenabilité. Toutefois, il est important de bien documenter les cas où une approche champ par champ est utilisée pour clarifier le traitement des exceptions.

### Conclusion

La gestion des variables dans les blocs tableaux de Sage X3, combinée avec les techniques de Trans-Classe et d'alimentation champ par champ, permet aux développeurs de construire des interfaces utilisateur dynamiques et efficaces, adaptées aux besoins spécifiques des entreprises. Cette approche offre un équilibre entre automatisation et contrôle personnalisé, essentiel pour le traitement précis et efficace des données dans les applications ERP.

# Gestion Avancée des Étiquettes, Sous-programmes et Fonctions dans Sage X3

Dans Sage X3, la structuration du code à travers l'utilisation d'étiquettes, de sous-programmes et de fonctions est essentielle pour une programmation modulaire et efficace. Cette approche permet non seulement une meilleure lisibilité du code, mais aussi une réutilisation optimale des blocs de logique métier à travers différents modules du système. Voici un aperçu détaillé de ces éléments, avec des exemples concrets pour illustrer leur mise en œuvre.

## Les Étiquettes (Labels)

Les étiquettes dans Sage X3 sont des parties de programme adressables par l'instruction Gosub, qui permettent un débranchement temporaire du flux principal pour exécuter une routine spécifique.

### Syntaxe et Exemple:

« # Déclaration de l'étiquette

$ACTION

# Corps de l'étiquette

Case ACTION

When "EXEC"

Gosub EXEC

Endcase

Return »

Dans cet exemple, $ACTION est une étiquette qui, selon la valeur de ACTION, peut exécuter différentes sous-routines. L'exécution retourne ensuite à la ligne suivant l'appel Gosub.

Appeler une étiquette dans Sage X3 se fait principalement à travers l'utilisation de l'instruction Gosub. Cette instruction permet de débrancher temporairement le flux d'exécution du programme principal pour exécuter un bloc de code spécifié par une étiquette, puis de revenir au point d'appel une fois ce bloc exécuté.

### Syntaxe de base de Gosub

La syntaxe pour appeler une étiquette avec Gosub est la suivante :

« Gosub NOM\_ETIQUETTE »

### Détails de l'implémentation

Définition de l'Étiquette: Avant de pouvoir appeler une étiquette, vous devez la définir quelque part dans votre code. Une étiquette est simplement un pointeur vers une section spécifique du code.

Marquer l'Étiquette: Vous marquez une étiquette en utilisant le symbole dollar ($) suivi du nom de l'étiquette, par exemple $NOM\_ETIQUETTE.

Écrire le Code de l'Étiquette: Après la déclaration de l'étiquette, vous écrivez le code que vous souhaitez exécuter quand l'étiquette est appelée.

Retour au Programme Principal: À la fin du bloc de code de l'étiquette, vous utilisez l'instruction Return pour indiquer que le contrôle doit retourner au point juste après l'appel de Gosub dans le programme principal.

### Exemple Complet

Voici un exemple complet qui illustre la définition, l'appel, et le retour d'une étiquette dans Sage X3 :

« # Main program starts here

Infbox "Début du programme principal"

# Call the label using Gosub

Gosub ACTION

# Continue with the rest of the main program

Infbox "Retour au programme principal et continuation"

End

# Label definition

$ACTION

# Code to execute when the label is called

Infbox "Action spécifique exécutée dans l'étiquette"

Return »

### Points Clés à Considérer

Clarté: Les étiquettes doivent être utilisées de manière claire et organisée pour éviter de rendre le code difficile à suivre, surtout dans les grands programmes.

Usage Limité: Préférez l'utilisation de sous-programmes et de fonctions pour une meilleure modularité et réutilisation du code, sauf dans les cas où un débranchement simple et rapide est nécessaire sans passer de paramètres.

Documentation: Commentez abondamment l'utilisation des étiquettes pour aider à la maintenance du code et pour clarifier leur rôle aux autres développeurs ou lors de la révision du code.

## Les Sous-programmes

Les sous-programmes dans Sage X3 permettent de regrouper des instructions dans des blocs qui peuvent être appelés avec des paramètres. Ces blocs facilitent la modularité et la gestion des erreurs en isolant des fonctionnalités spécifiques.

### Syntaxe et Exemple:

« # Déclaration du sous-programme

Subprog ZDAT (DAT,ZERR)

Value date DAT

Variable integer ZERR

If DAT > date$ : ZERR = 1 : Endif

End

# Appel du sous-programme

Local integer ZERR

Call ZDAT ([31/12/2005],ZERR) From TRTXXX »

Dans cet exemple, ZDAT est un sous-programme qui prend deux paramètres: une date DAT et une variable entière ZERR qui retournera un code d'erreur. Call est utilisé pour invoquer ce sous-programme, en spécifiant les valeurs des paramètres et le contexte du traitement (TRTXXX).

## Les Fonctions

Les fonctions dans Sage X3, comme les sous-programmes, sont des blocs de code qui retournent une valeur. Elles sont particulièrement utiles pour effectuer des calculs ou des opérations qui nécessitent une valeur de retour.

Syntaxe et Exemple:

« # Déclaration de la fonction

Funprog CHAR\_(CLOB)

Value Clbfile CLOB

Local Char CHAINE(255)

Setlob CHAINE With CLOB

End CHAINE

# Appel de la fonction

Local Char RESULT(255)

RESULT = Func CHAR\_([F:TXC]TEXTE)  »

Cet exemple montre comment une fonction CHAR\_ est définie pour convertir un objet CLOB en une chaîne de caractères. La fonction est appelée avec Func, et le résultat est stocké dans la variable RESULT.

## Best Practices

Modularité: Utilisez des sous-programmes et des fonctions pour encapsuler des comportements qui sont répétés à travers différentes parties de l'application. Cela réduit la redondance du code et centralise la logique pour faciliter les mises à jour et la maintenance.

Gestion des erreurs: Implémentez une gestion d'erreurs robuste dans vos sous-programmes et fonctions pour traiter les exceptions de manière cohérente.

Performance: Évitez des appels de fonctions ou sous-programmes excessifs dans des boucles, car cela peut impacter la performance. Évaluez l'efficacité de vos routines en termes de temps d'exécution et d'utilisation des ressources.

## Conclusion

L'usage judicieux des étiquettes, sous-programmes et fonctions dans Sage X3 non seulement structure le code de manière efficace mais assure aussi que les applications sont robustes, maintenables et performantes. Chaque élément a un rôle spécifique à jouer dans la conception des logiciels, facilitant ainsi la création de solutions complexes qui répondent efficacement aux exigences métier.

# REGLES IMPORTANTES D’OPTIMISATION DES REQUETES

L'optimisation des requêtes est une composante cruciale du développement efficace dans Sage X3, permettant de maximiser la performance et de minimiser la charge sur la base de données. Appliquer des stratégies d'optimisation adéquates peut grandement influencer la réactivité et la stabilité de l'application. Voici une exploration détaillée des règles de base pour l'optimisation des requêtes dans Sage X3.

## Minimiser le Nombre de Requêtes

### Utilisation de Link vs. Multiples Lectures :

Principe : Privilégier l'usage de Link pour effectuer des jointures directement sur le serveur de base de données plutôt que de réaliser plusieurs lectures séquentielles (Read). Cela réduit le trafic réseau et le nombre de requêtes SQL exécutées.

Exemple : Utiliser Link pour associer des données de plusieurs tables en une seule opération.

### Boucles For vs. Séries de Read :

Principe : Favoriser les boucles For pour parcourir des enregistrements, car elles génèrent moins d'instructions SQL que des appels multiples à Read dans une boucle.

Exemple : Remplacer les multiples Read dans une boucle While ou Repeat par une boucle For structurée.

### Optimisation des Conditions dans les Requêtes

Éviter les opérateurs OR :

Principe : Convertir des conditions utilisant OR en une forme utilisant AND et des comparaisons de plages, si possible, pour une exécution plus rapide et plus efficace.

Exemple : Remplacer For [XXX] Where ENTIER=3 or ENTIER=4 or ENTIER=5 par For [XXX] Where ENTIER>=3 and ENTIER<=5.

### Limiter les Données Traitées :

Principe : Utiliser Columns pour spécifier les colonnes à récupérer avant une boucle For, évitant ainsi le coûteux Select \* From .... Utiliser Update pour les mises à jour batch sur des champs spécifiques, plutôt que de multiples opérations de Rewrite.

Exemple : Mettre à jour les champs spécifiques d'un grand nombre d'enregistrements avec une instruction Update plutôt que de multiples Rewrite.

## Gestion Raisonnée des Mises à Jour

### Groupage des Opérations :

Principe : Regrouper raisonnablement les mises à jour et les lectures pour éviter les erreurs telles que "snapshot too old" sous Oracle, tout en évitant une charge excessive en termes de nombre de requêtes.

Exemple : Regrouper les mises à jour en transactions logiques qui ne sont ni trop grandes pour éviter les blocages, ni trop petites pour prévenir une surcharge de requêtes.

Filtrage Efficace et Fonctions SQL

### Optimisation du Filtrage :

Principe : Utiliser des fonctions et des opérateurs qui sont efficacement traduits en SQL, tels que >,>=,<,<=,<>,=, ainsi que len, ascii, chr$, toupper, tolower.

Cas particuliers : Utiliser Find et pat dans les clauses Where pour effectuer des recherches optimisées. Par exemple, For [XXX] Where pat(CHAMP, "\*MODELE\*") <>0 est traduit en SQL par Select ... Where CHAMP like '%MODELE%'.

### Conclusion

L'optimisation des requêtes dans Sage X3 nécessite une compréhension approfondie des capacités du système et des meilleures pratiques en matière de manipulation des données SQL. En suivant ces principes, les développeurs peuvent non seulement améliorer la performance des applications mais aussi garantir leur fiabilité et leur efficacité à long terme.

# L’utilisation du Transactionnel pour les modifications de Tables en L4G

## Synthèse sur l'Utilisation de Trbegin, Rollback, et Commit dans Sage X3

L'emploi correct des commandes Trbegin, Rollback, et Commit dans Sage X3 est essentiel pour assurer l'intégrité des données et la gestion efficace des transactions dans les applications ERP. Ces commandes permettent de contrôler et sécuriser les modifications apportées aux données, en particulier dans des environnements où les opérations sont complexes et interdépendantes. Voici une synthèse générale sur leur utilisation, en mettant l'accent sur les bonnes pratiques et les précautions à prendre.

### Préparation avant la Transaction

Avant de commencer une transaction avec Trbegin, il est crucial de s'assurer que toutes les tables impliquées dans la transaction sont ouvertes. Ceci est nécessaire car une transaction peut impliquer plusieurs opérations sur différentes tables, et l'ouverture préalable garantit que toutes les ressources nécessaires sont disponibles et prêtes à être utilisées.

### Démarrage de la Transaction

La commande Trbegin initie une transaction. Il est important de noter que Trbegin ne doit jamais être utilisé seul ; il doit toujours spécifier les tables qui seront touchées par la transaction. Cela permet de clarifier le contexte de la transaction et de s'assurer que toutes les modifications ultérieures se font dans un environnement contrôlé.

### Exemple de bonne pratique :

plaintext

Copy code

Trbegin [F:Table1], [F:Table2]

### Suivi et Gestion des Erreurs

L'utilisation d'une variable, typiquement nommée WERR (pour "Workflow ERRor"), est une pratique courante pour suivre l'état d'une transaction. Cette variable aide à identifier les erreurs qui peuvent survenir à différents points du processus de transaction. Le contrôle du statut des opérations (fstat) doit être effectué immédiatement après chaque opération critique telle que Read, Filter, Write, Rewrite, ou Delete. Ceci est impératif pour s'assurer que chaque étape de la transaction est exécutée sans erreur avant de passer à la suivante.

### Exemple de contrôle de fstat :

plaintext

Copy code

Read [F:Table]Key = Value

If [S]fstat <> 0 Then

WERR = 1 # Une erreur s'est produite

Endif

### Annulation et Validation de la Transaction

Si à un moment donné, WERR indique une erreur (i.e., WERR est différent de zéro), la transaction doit être annulée pour éviter des modifications partielles ou incorrectes dans la base de données. Ceci est réalisé avec la commande Rollback, qui annule toutes les modifications effectuées depuis le début de la transaction.

En revanche, si la transaction s'est déroulée sans erreurs (i.e., WERR est égal à zéro), elle peut être validée et rendue permanente avec la commande Commit.

### Exemple de gestion de fin de transaction :

plaintext

Copy code

If WERR = 0 Then

Commit # Valide la transaction si aucune erreur

Else

Rollback # Annule la transaction en cas d'erreur

Endif

### Conclusion

L'utilisation judicieuse de Trbegin, Rollback, et Commit est essentielle pour gérer de manière sécurisée les transactions dans Sage X3. Il est crucial que les développeurs suivent les pratiques recommandées pour éviter les erreurs de transaction, maintenir l'intégrité des données, et optimiser la performance de l'application. En respectant ces principes, les développeurs peuvent assurer que les modifications des données sont gérées de manière fiable et efficace.

## Analyse et Explication d'un Script de Gestion Transactionnelle dans Sage X3

Le script présenté ci-dessous illustre un processus transactionnel typique dans Sage X3, où plusieurs tables sont manipulées pour effectuer des mises à jour basées sur des critères spécifiques. Cet exemple met en évidence la gestion des erreurs, l'utilisation stratégique des transactions, et l'importance de la vérification du statut après chaque opération de base de données.

### Ouverture des Tables et Initialisation des Variables

Avant toute manipulation, le script s'assure que toutes les tables nécessaires sont ouvertes. Cette étape est cruciale car tenter d'accéder à une table non ouverte entraînerait une erreur.

plaintext

Copy code

If clalev([F:ZRCL])=0 : Local File ZRECUPCLT [F:ZRCL] : Endif

If clalev([F:ZDPO])=0 : Local File ZDEPOT [F:ZDPO] : Endif

If clalev([F:ZAI])=0 : Local File ZADREPI [F:ZAI] : Endif

If clalev([F:ZAI2])=0 : Local File ZADREPI [F:ZAI2] : Endif

Ces lignes garantissent que chaque table est accessible pour les opérations suivantes, et chaque table est ouverte seulement si elle n'a pas déjà été ouverte (clalev vérifie le niveau d'accès actuel).

### Début de la Transaction

La transaction est initiée avec Trbegin et inclut toutes les tables qui seront modifiées. Cela prépare le système à gérer toutes les modifications de manière atomique, c'est-à-dire que toutes les modifications seront appliquées ou aucune ne le sera si une erreur survient.

plaintext

Copy code

Trbegin [F:ZDPO],[F:ZAI],[F:ZAI2]

Il est important de noter que le Trbegin inclut les tables en argument pour spécifier explicitement les tables impliquées dans la transaction, conformément aux meilleures pratiques.

### Traitement des Enregistrements et Gestion des Erreurs

Le script procède à des itérations sur des enregistrements de table, effectuant des opérations de lecture, écriture et mise à jour. Chaque opération est suivie d'une vérification de fstat, la variable système qui indique le résultat de la dernière opération de base de données.

plaintext

Copy code

For[F:ZRCL]ZRCL0 Where [F:ZRCL]ZCODCLT=[F:ZRCL]ZCODCLTREG

...

Write[F:ZDPO]

If(fstat=0)Then

...

Rewrite[F:ZAI]

If(fstat<>0)Then

WERR = 1

Endif

Else

WERR = 3

Endif

Next

Dans ce bloc, WERR est utilisé comme indicateur d'erreur pour suivre les problèmes rencontrés pendant la transaction. Si une erreur est détectée (fstat n'est pas zéro), WERR est mis à jour avec un code spécifique indiquant le type d'erreur.

### Clôture et Validation de la Transaction

À la fin du script, le processus vérifie WERR pour décider de valider (Commit) ou d'annuler (Rollback) la transaction.

plaintext

Copy code

If(WERR=0)Then

Commit

Call MESSAGE("Traitement terminé avec succès") From GESECRAN

Else

Rollback

Call ERREUR("Traitement annulé car il existe des erreurs. [Erreur n°" + num$(WERR) + "]") From GESECRAN

Endif

Si WERR est zéro, cela signifie que toutes les opérations se sont déroulées sans erreur, et les modifications peuvent être validées de manière permanente dans la base de données. En cas d'erreur, toutes les modifications sont annulées pour préserver l'intégrité des données.

## Conclusion

Ce script démontre l'importance d'une gestion rigoureuse des transactions dans Sage X3 pour assurer la cohérence et la fiabilité des données. Les développeurs doivent toujours s'assurer que toutes les tables impliquées dans une transaction sont spécifiées dans Trbegin, que chaque opération de base de données est suivie par une vérification de fstat, et que les transactions sont correctement clôturées en fonction des résultats de ces vérifications. En suivant ces principes, les développeurs peuvent minimiser les erreurs et garantir que les processus d'affaires sont exécutés de manière fiable et sécurisée.

# Gestion des Traces dans Sage X3

La gestion des traces est une pratique essentielle dans le développement et la maintenance des systèmes informatiques, notamment dans Sage X3. Les traces permettent de suivre les opérations effectuées par le système, offrant ainsi un aperçu détaillé des processus en cours, ce qui est crucial pour le débogage et l'optimisation des performances. Voici une explication détaillée de la gestion des traces dans Sage X3, illustrée par des exemples.

## Concepts de Base

### Ouverture et Fermeture du Fichier Trace

Les fichiers de trace sont utilisés pour enregistrer les événements et les opérations effectuées durant l'exécution des traitements. L'ouverture d'un fichier de trace se fait souvent au début d'un traitement et il est fermé à la fin pour s'assurer que toutes les informations sont correctement sauvegardées et que les ressources sont libérées.

### Exemple d'ouverture et de fermeture :

« Local char TIT(30) : [L]TIT = ‘Formation langage’

If !GSERVEUR : Call OUVRE\_TRACE(TIT) From LECFIC : Endif

…

If !GSERVEUR : Call FERME\_TRACE From LECFIC : Endif »

### Lecture et Écriture dans la Trace

Lecture : Les traces enregistrées peuvent être relues pour l'analyse. Cela est souvent réalisé via un traitement spécifique qui permet de parcourir le contenu d'un fichier trace.

« TRA=1

Call LEC\_TRACE from LECFIC »

Écriture : L'écriture dans les fichiers trace se fait via des appels spécifiques où l'on spécifie le message et le type de message (informatif, erreur, avertissement).

« Call ECR\_TRACE(‘Bonjour’, 0) from GESECRAN

Call ECR\_TRACE(‘Erreur:"- num$(NUERR), 1) From GESECRAN »

### Gestion Avancée des Traces

Formatage des Messages dans les Traces

Les messages écrits dans les fichiers trace peuvent inclure des chemins d'accès ou d'autres informations nécessitant un formatage spécial pour éviter des interprétations erronées par le système.

Pour écrire un chemin dans une trace tout en évitant des erreurs de formatage dues au caractère \, utilisez | pour délimiter le chemin.

« Call ECR\_TRACE("|c:\temp\monfic.txt|") From GESECRAN »

### Nommage des Fichiers Trace

Les fichiers de trace sont généralement stockés dans un répertoire spécifique et suivent un modèle de nommage qui facilite leur identification et leur gestion.

Emplacement et nommage : Les fichiers sont souvent nommés sous la forme RQTnnn.tra, où nnn représente le numéro de la requête batch. Ils sont stockés dans le répertoire ..\runtime\SERVX3\TRA.

Exemple Générique de Gestion des Traces

« # Initialisation du titre de la trace

Local char TIT(30) : [L]TIT = ‘Début du traitement des commandes’

# Ouverture du fichier de trace si en mode serveur

If !GSERVEUR : Call OUVRE\_TRACE(TIT) From LECFIC : Endif

# Traitement principal ici

...

# Ecriture d'une information dans la trace

Call ECR\_TRACE("Traitement des commandes terminé", 0) From GESECRAN

# Fermeture du fichier de trace après le traitement

If !GSERVEUR : Call FERME\_TRACE From LECFIC : Endif »

### Conclusion

La gestion des traces dans Sage X3 est un outil puissant pour le diagnostic et l'analyse des processus d'application. En utilisant efficacement les fonctions de trace, les développeurs peuvent obtenir des informations détaillées sur l'exécution des programmes, facilitant ainsi le débogage et l'optimisation des performances. Il est crucial de bien structurer et documenter les mécanismes de trace pour maintenir la clarté et l'efficacité du système.

# Les fonctions L4G de manipulations de variables

Voici un tableau récapitulatif des fonctions disponibles dans le langage L4G de Sage X3, couvrant les fonctions pour les chaînes de caractères, les dates, les expressions numériques et diverses autres fonctions utiles. Ce tableau fournit une description concise de chaque fonction pour faciliter leur compréhension et leur application.

## Fonctions pour les Chaînes de Caractères

| Fonction | Description |
| --- | --- |
| left$(chaîne, inb) | Extrait les inb premiers caractères de chaîne. |
| right$(chaîne, ipos) | Extrait les caractères de chaîne à partir de ipos. |
| mid$(chaîne, ipos, inb) | Extrait les inb caractères de chaîne à partir de ipos. |
| seg$(chaîne, ideb, ifin) | Extrait la sous-chaîne comprise entre ideb et ifin. |
| len(chaîne) | Retourne la longueur de chaîne. |
| val(chaîne) | Convertit une chaîne en valeur numérique. |
| num$(nombre) | Convertit un nombre en chaîne de caractères. |
| tolower(chaîne) | Convertit une chaîne en minuscules. |
| toupper(chaîne) | Convertit une chaîne en majuscules. |
| ctrans(chaîne [, ci, co]) | Filtre les caractères de chaîne. |
| vireblc(chaîne, ioption) | Supprime les blancs dans une chaîne. |
| format$(cfmt, chaîne) | Formatage de chaîne avec cfmt. |
| pat(chaîne, cmodèle) | Correspondance de chaîne avec le modèle cmodèle. |
| instr(ipos, chaîne, cssch) | Recherche cssch dans chaîne à partir de ipos. |
| space$(inb) | Génère une chaîne constituée de inb blancs. |
| string$(inb, chaîne) | Répète chaîne inb fois. |
| ascii(chaîne) | Retourne le code ASCII du premier caractère de chaîne. |
| chr$(iascii) | Retourne le caractère dont le code ASCII est iascii. |
| parse(chaîne [, btoken]) | Analyse syntaxique de chaîne. |
| evalue(chaîne [, btoken]) | Évalue l'expression contenue dans chaîne. |
| getenv$(chaîne) | Retourne la valeur de la variable système définie dans chaîne. |

## Fonctions pour les Dates

| Fonction | Description |
| --- | --- |
| date$ | Retourne la date du jour (Serveur). |
| time$ | Retourne l’heure courante sous la forme HH:MM:SS. |
| time | Heure courante en secondes. |
| day(date) | Extrait le jour d’une date. |
| day$(date) | Retourne le nom du jour d’une date. |
| month(date) | Extrait le mois d’une date. |
| month$(date) | Retourne le nom du mois d’une date. |
| year(date) | Extrait l’année d’une date. |
| gdat$(ijour, imois, iannée) | Retourne la date correspondant à [ijour/imois/iannée]. |
| nday(date) | Retourne le nombre de jours depuis le 1/1/1600 de date. |
| nday$(inbjour) | Convertit un nombre de jours depuis le 1/1/1600 en date. |
| dayn(date) | Retourne le numéro de jour dans la semaine de date. |
| week(date) | Retourne le numéro de semaine d’une date. |
| aweek(isemaine, iannée) | Date du début de la semaine numéro isemaine de iannée. |
| eomonth(date) | Retourne la date de la fin du mois d’une date. |
| addmonth(date, inb) | Ajoute inb mois à une date. |
| format$ | Mise en forme d’une date. |

## Fonctions pour les Expressions Numériques

| Fonction | Description |
| --- | --- |
| abs(nombre) | Valeur absolue de nombre. |
| mod(ientier, idiv) | Modulo entre ientier et idiv. |
| rnd(nombre) | Génère un nombre aléatoire entre 0 et nombre. |
| sgn(nombre) | Retourne le signe de nombre. |
| sqr(nombre) | Calcule la racine carrée de nombre. |
| ar2(nombre) | Arrondi de nombre à 2 décimales. |
| arr(nombre, nprec) | Arrondi de nombre à nprec décimales près. |
| fix(nombre) | Troncature de nombre. |
| int(nombre) | Partie entière de nombre. |
| fac(ientier) | Factorielle de ientier. |
| anp(in, ip) | Arrangements de ip dans in. |
| cnp(in, ip) | Combinaisons de ip dans in. |
| Format$ | Mise en forme d’un nombre. |

## Fonctions Diverses

| Fonction | Description |
| --- | --- |
| find(valeur, liste) | Recherche d’une valeur dans une liste. |
| uni(liste) | Retourne l'indice du premier doublon dans la liste. |
| min(liste) | Retourne la valeur minimale d’une liste. |
| max(liste) | Retourne la valeur maximale d’une liste. |
| avg(liste) | Calcule la moyenne arithmétique d’une liste. |
| var(liste) | Calcule la variance d’une liste. |
| sum(liste) | Calcule la somme des éléments d’une liste. |
| prd(liste) | Calcule le produit des éléments d’une liste. |
| sigma([var =] ideb, ifin, expr) | Réalise une sommation de expr avec var variant entre ideb et ifin. |
| [S]indcum | Variable de boucle par défaut de sigma(). |

# Manipulation des Écrans dans Sage X3

Dans Sage X3, la manipulation des écrans est une composante essentielle de la gestion de l'interface utilisateur. Ce chapitre détaille les instructions spécifiques utilisées pour contrôler les écrans, notamment l'ouverture, la fermeture, l'affichage, le grisage et le dégrisage des zones, ainsi que la gestion de la validation des écrans. Des exemples pratiques sont fournis pour illustrer chaque instruction.

## Ouverture et Fermeture d’un Écran

Classe [M] : Chaque écran dans Sage X3 est géré comme une classe de variables, où [M] indique une classe d'écran.

### Ouverture d'un écran :

Syntaxe : [Local] Mask <idécran> [<classe>]

Utilisation : Ouvre un écran en précisant éventuellement l'abréviation de la classe.

Exemple :

« If clalev([M:SOH0]) = 0 : Local Mask SOH0 [SOH0] : Endif »

Cette instruction vérifie si l'écran SOH0 est déjà ouvert ; si ce n'est pas le cas, il l'ouvre localement.

### Fermeture d'un écran :

Syntaxe :

« Close [Local] Mask [<liste\_classe>] »

Ferme un ou plusieurs écrans ouverts.

## Test d'Accessibilité d'une Classe

Instruction clalev(<classe>) :

Utilisation : Permet de tester si une classe de variables est accessible par un programme.

Exemple :

« If clalev([M:SOH0]) = 0 : Local Mask SOH0 [SOH0] : Endif »

## Affichage des Zones (AFFZO)

Utilisation : Permet d'afficher les zones d'un masque ouvert avec des restrictions éventuelles.

Exemples :

« Affzo [M:BPC1]

Affzo [M:BPC1] BPCGRU, BPCBPSNUM

Affzo [M:BPC1] 1-99 »

## Effacement des Zones (EFFZO)

Utilisation : Efface les zones d'un masque ouvert et les considère comme non initialisées.

Exemple :

« Effzo [M:BPC1] BPCGRU »

## Grisage des Zones (GRIZO)

Utilisation : Rend les zones spécifiées non saisissables et visuellement grises.

Exemple :

« Grizo [M:SOH0] SOHNUM »

## Dégrisage des Zones (ACTZO)

Utilisation : Réactive les zones spécifiées, les rendant à nouveau saisissables.

Exemple :

« Actzo [M:SOH0] SOHNUM »

## Validation d'un Écran (SETMOK)

Utilisation : Agit sur le statut validé de l'écran.

Exemple :

« Setmok [M:BPC1] with 1 # Rend l'écran valide »

## Suppression de Lignes (DELA)

Utilisation : Supprime une ou plusieurs lignes d'un tableau à partir d'un indice spécifié.

Exemple :

« Dela 5,1 [M:NBLIG] »

## Insertion de Lignes (INSA)

Utilisation : Insère une ou plusieurs nouvelles lignes dans un tableau à un indice spécifié.

Exemple :

« Insa 10,1 [M:NBLIG] »

### Conclusion

La manipulation des écrans dans Sage X3 offre une flexibilité considérable dans la gestion de l'interface utilisateur. Chaque instruction a un rôle spécifique qui permet de contrôler finement l'affichage et l'interaction avec les écrans. Les développeurs doivent maîtriser ces instructions pour créer des interfaces utilisateur efficaces et réactives, garantissant ainsi une expérience utilisateur optimale et conforme aux processus métiers de l'organisation.

## Analyse d’un script complet de manipulation d’écran Sage X3

Nous allons analyser le script suivant :

« ###########################################################################

# TRT\_LIGCDE Détermination de la quantité à livrer #

# et alimentation de la ligne de livraison #

#-------------------------------------------------------------------------#

# Utilisation : picking de commandes depuis la gestion livraison #

# bouton livraison depuis la gestion commandes #

###########################################################################

$TRT\_LIGCDE

Local Integer WRUP : # Génération des ruptures (0=non / 1=oui)

Local Integer WRET

Local Integer WSTA, WA, WQ, WR

Local Integer WSTOSEQ

Local Decimal WQTY, WQTYSTU, WSTUDIS, WQTYARR, WQTA

Local Decimal WQTYCPE, WQTYSTUCPE, WCOECPE, WQTYDLVCPE

Local Char WSAU(GLONUOM)

Local Decimal WSAUSTUCOE

Local Decimal WOALQTYSTU : # Qté allouée origine en US

Local Decimal WOALQTYCPE : # Qté allouée origine du composé en US

Local Decimal WSVGQTYSTU

Local Integer WNEGSTO

WNEGSTO = [F:ITM]NEGSTO

GPOINT = "TRT\_LIGCDE" : Gosub ENTREE From EXEFNC

WSAUSTUCOE = 1

WSAU = [F:ZPD]SAU

If WSAUSTUCOE=0 WSAUSTUCOE=1 Endif

#----------------------------------------

# Détermination de la quantité à livrer

#----------------------------------------

#WQTY = [F:ZPD]QTY-[F:ZPD]QTYCDE-[F:ZPD]QTYLIV

WQTY = [F:ZPD]QTY-[F:ZPD]QTYCDE

WQTYSTU = WQTY

Raz WRUP

# Relecture des tables articles (Bug 42512)

Gosub LEC\_TABART\_EPI

#--- La sélection est correcte

# Gosub SOH\_SDH\_PLUS From SUBSFIT

# Sauvegarde du lien entre ligne document et écran travail

[L]WSTOSEQ = [M:SDH1]WSTOSEQ([M:SDH1]NBLIG)

# Ouverture de la ligne

nolign = [M:SDH1]NBLIG

Insa nolign,1,[M:SDH1]NBLIG [M:SDH1]NBLIG

[M:SDH1]NBLIG += 1

nolign = [M:SDH1]NBLIG

NOL = nolign-1

status = 0

Call LINNUM("SDDLIN") From TABLEAUX

Call ALICLISDH ("C",[M:SDH0]BPCORD) From TRTVENLIV

# Alimentation du 1er représentant

If dim([M:SDH2]REP) > 0

[M:SDH2]REP(0) = [F:BPC]REP(0)

Endif

# Alimentation du 2eme représentant

If dim([M:SDH2]REP) > 1

If [M:SDH2]REP(0) <> ""

[M:SDH2]REP(1) = [F:BPC]REP(1)

Endif

Endif

[M:SDH1]ITMREF(NOL) = [F:ZPD]ITMREF

#On va maintenant récupérer le prix de al relocalisation

#[M:SDH1]GROPRI(NOL) = func ZWSEPALIS.GET\_TARIF\_RELOC([F:ZPD]ITMREF,[M:SDH0]BPCORD,[M:SDH0]BPAADD,[M:SDH0]STOFCY,[F:ZPD]QTY-[F:ZPD]QTYCDE-[F:ZPD]QTYLIV)

[M:SDH1]GROPRI(NOL) = func ZWSEPALIS.GET\_TARIF\_RELOC([F:ZPD]ITMREF,[M:SDH0]BPCORD,[M:SDH0]BPAADD,[M:SDH0]STOFCY,[F:ZPD]QTY-[F:ZPD]QTYCDE,"","",[M:SDH1]ZTYPPAL(NOL),"","",date$)

[M:SDH1]ZPDLIN(NOL) = [F:ZPD]ZPDLIN

[M:SDH1]ZPSNUM(NOL) = [F:ZPD]ZPSNUM

[M:SDH1]LINTYP(NOL) = 1

[M:SDH1]ITMREF(NOL) = [F:ZPD]ITMREF

[M:SDH1]ITMDES(NOL) = [F:ZPD]ITMDES1

[M:SDH1]QTY(NOL) = WQTY

[M:SDH1]QTYSTU(NOL) = WQTYSTU

[M:SDH1]VCRTYPORI(NOL) = 4

[M:SDH1]REPCOE(NOL) = 1

[M:SDH1]PRIREN(NOL) = 1

[M:SDH1]CCE1(NOL) = [F:ZPS]ZAXE1

[M:SDH1]CCE2(NOL) = [F:ZPS]ZAXE2

#On met à jour les axes analytiques

[M:SDH3]CCE(0) = [F:ZPS]ZAXE1

[M:SDH3]CCE(1) = [F:ZPS]ZAXE2

Affzo [M:SDH3]

Call ALIARTSDH ([F:ZPD]ITMREF,NOL) From TRTVENLIV

# Alimentation de la classe [M:SDH1] de la ligne

#Call ALISOQSDH(NOL,WOALQTYSTU,WQTY,WQTYSTU) From TRTVENLIV

# Récupération du lien entre ligne document et écran travail

[M:SDH1]WSTOSEQ(NOL) = [L]WSTOSEQ

# Calculs prix net, marge et maj montants livrés et poids

GNETMAR=0

Call ALICLCAMT([M:SDH1]ITMREF(NOL),[M:SDH1]QTY(NOL),NOL,"SDH1",[M:SDH1]CLCAMT1(NOL),

& [M:SDH1]CLCAMT2(NOL)) From TRTX3

# Bug 59754:Calcul prix net ssi il y a eu une recherche tarif sinon il faut garder le prix net de la cde

# (Si frais/remise dépendant de la qté (par ligne) et plusieurs liv.,le prix net doit être le même sur chaque liv.)

#If [F:SOH]SOHCAT=4 & ([F:SOH]BETFCY=2 | [F:SOC]PLI<>"")

Call CLCNETPRI([M:SDH1]QTY(NOL),[M:SDH1]CUR, NOL)From TRTVENPRI

#Endif

Call CLCPFM([M:SDH0]STOFCY,[M:SDH1]PRITYP,[M:SDH0]CHGTYP,[M:SDH1]SHIDAT,

& [M:SDH1]CUR,NOL,1) From TRTVENPRI

Call APRES\_LIGNE(NOL,0) From SUBSDHB

#--CPO 79343

If GIMPORT=0 and GREP<>"C" and GSDHSUPPLIGDSP<>999999

Local Integer WTEMPDSP : Raz WTEMPDSP

Call CTLSUPP\_EXISTDSP("SDH",NOL,WTEMPDSP) From TRTVENDIV

If WTEMPDSP=1

GSDHSUPPLIGDSP +=1

Endif

Endif

#--/CPO

GNETMAR=1

Raz GOUVSOR

nolign = NOL+1

# Sauvegarde qté cde et qté à livrer du composé

WQTYCPE = [F:SOQ]QTY

WALLQTYCPE = WQTY

WQTYDLVCPE = WQTY

Return »

Le script intitulé TRT\_LIGCDE est un exemple détaillé de manipulation de données dans Sage X3, spécifiquement pour le processus de livraison et de gestion des commandes. Il illustre plusieurs concepts clés de la programmation en L4G de Sage X3, notamment la gestion des écrans, les transactions, les appels de sous-programmes, et la manipulation des données au sein des masques. Analysons ce script en sections pour mieux comprendre son fonctionnement.

### Préparation et Initialisation

Avant d'entrer dans le cœur du traitement, le script initialise des variables locales pour gérer les états et les quantités tout au long du processus.

« Local Integer WRUP, WRET, WSTA, WA, WQ, WR, WSTOSEQ

Local Decimal WQTY, WQTYSTU, WSTUDIS, WQTYARR, WQTA

Local Decimal WQTYCPE, WQTYSTUCPE, WCOECPE, WQTYDLVCPE

Local Char WSAU(GLONUOM)

Local Decimal WSAUSTUCOE

Local Decimal WOALQTYSTU, WOALQTYCPE

Local Decimal WSVGQTYSTU

Local Integer WNEGSTO »

Ces variables incluent des indicateurs de gestion de rupture (WRUP), des compteurs et divers indicateurs pour le traitement des lignes de commande.

### Lecture des Données Initiales

Le script commence par lire des valeurs directement depuis les tables, souvent nécessaires pour des vérifications conditionnelles ou des calculs initiaux.

« WNEGSTO = [F:ITM]NEGSTO »

Cette ligne charge dans WNEGSTO un indicateur spécifique de la table des articles, utile pour des contrôles ultérieurs.

### Début du Traitement

Une sous-routine est appelée pour préparer le traitement.

« GPOINT = "TRT\_LIGCDE"

Gosub ENTREE From EXEFNC »

Cela indique l'entrée dans une section spécifique de traitement, dénommée ici TRT\_LIGCDE.

### Détermination de la Quantité à Livrer

Le script calcule la quantité restante à livrer après soustraction des quantités déjà commandées et livrées.

« WQTY = [F:ZPD]QTY - [F:ZPD]QTYCDE

WQTYSTU = WQTY »

### Manipulation des Écrans et Lignes

La sauvegarde des liens entre les lignes de documents et l'écran de travail est cruciale pour le bon fonctionnement des interfaces utilisateur.

« [L]WSTOSEQ = [M:SDH1]WSTOSEQ([M:SDH1]NBLIG) »

Cette ligne sauvegarde un identifiant de séquence pour référence ultérieure dans la session active.

### Insertion de Lignes et Mise à Jour des Masques

Le script insère des lignes dans un écran/masque et met à jour les champs associés.

« Insa nolign, 1, [M:SDH1]NBLIG

[M:SDH1]NBLIG += 1 »

Cela insère une nouvelle ligne dans le masque SDH1 et met à jour le compteur de ligne.

### Appels de Fonctions et Mise à Jour des Données

Des fonctions externes sont appelées pour calculer des tarifs ou mettre à jour des informations spécifiques liées aux articles.

« [M:SDH1]GROPRI(NOL) = func ZWSEPALIS.GET\_TARIF\_RELOC([F:ZPD]ITMREF, ...) »

Cette instruction calcule le tarif de relocalisation pour un article spécifique.

### Validation et Sauvegarde

Enfin, le script gère la validation et la sauvegarde des modifications apportées lors du processus.

« If GIMPORT = 0 and GREP <> "C" and GSDHSUPPLIGDSP <> 999999

Call CTLSUPP\_EXISTDSP("SDH", NOL, WTEMPDSP) From TRTVENDIV »

Cela vérifie certaines conditions avant de procéder à d'autres appels de fonction ou de validations.

### Conclusion

Ce script est un exemple typique de la façon dont les processus complexes sont gérés dans Sage X3. Il montre la nécessité de bien comprendre la logique métier, les structures de données, et la manipulation des écrans pour développer des solutions efficaces et robustes. L'intégration des différents éléments de données, la manipulation des interfaces utilisateur, et la logique transactionnelle doivent être méticuleusement planifiées pour assurer l'exactitude et la performance du système.

# Gestion des Boîtes de Dialogue dans Sage X3

La gestion des boîtes de dialogue dans Sage X3 est essentielle pour fournir une interaction efficace et intuitive avec l'utilisateur final. Sage X3 offre une variété de méthodes pour afficher des messages, poser des questions, ou donner des avertissements, enrichissant ainsi l'interface utilisateur. Cet article explore les différentes options disponibles pour la gestion des boîtes de dialogue dans Sage X3, en soulignant les meilleures pratiques et en fournissant des exemples concrets.

## Méthodes Simples de Message

### Infbox:

Infbox est la méthode la plus simple pour afficher un message.

Usage: Elle est généralement utilisée pour des messages rapides et informatifs.

Exemple:

« Infbox "Bonjour" »

### Message:

Message est une autre fonction simple pour afficher des messages.

Avantage: Contrairement à Infbox, si exécutée en mode batch, Message gère l'écriture du message dans la trace automatiquement; un message Infbox serait perdu.

Exemple:

« call MESSAGE("Bonjour") from GESECRAN »

## Dialogues Interactifs

### Boîtes Oui/Non (Yes/No boxes):

Utilisées pour demander une confirmation à l'utilisateur.

Syntaxe:

« Local Integer YN

YN = 2 # Par défaut pointe sur Oui

Call OUINON("Do you want to continue?", YN) From GESECRAN »

Exemple d'utilisation:

« If YN=1 : Infbox "YOU CLICKED NO" : Endif

If YN=2 : Infbox "YOU CLICKED YES" : Endif »

### Boîtes OK/Annuler (Ok/Cancel boxes):

Utilisées pour avertir l'utilisateur avant des actions critiques.

Syntaxe:

« Local Integer OKCAN

OKCAN = 2 # Ok par défaut

Call AVERTIR("Warning: All records will be deleted", OKCAN) From GESECRAN »

Exemple d'utilisation:

« If OKCAN=1 : Infbox "YOU CLICKED OK" : Endif

If OKCAN=2 : Infbox "YOU CLICKED Cancel" : Endif »

## Gestion des Erreurs et Avertissements

### Messages d'Erreur et d'Information:

ERREURT permet d'afficher des messages d'erreur ou d'information avec un paramètre pour spécifier le type.

Exemple:

« Call ERREURT("Erreur dans l'insertion", 1) From GESECRAN # 1 pour erreur

Call ERREURT("Information sur le processus", 0) From GESECRAN # 0 pour information »

## Messages Progressifs et d'Attente

### Message Progressif (Progressive waiting message):

Pour indiquer l'avancement d'une opération prolongée.

Syntaxe:

« Call TEMPON("Action en cours...0%") From GESECRAN

# Suite des instructions

Call TEMPON("Action en cours...50%") From GESECRAN

# Fin des instructions

Call TEMPOFF From GESECRAN »

## Personnalisation et Paramétrage

### Multi-ligne et Timeout:

Les messages peuvent inclure des textes sur plusieurs lignes en utilisant \ comme séparateur.

Un timeout peut être défini pour que la fenêtre se ferme automatiquement après un certain temps.

Exemple avec timeout:

« GTIMOUT=15 # timeout de 15 secondes

Call ERRTITT("Erreur", "Titre personnalisé") From GESECRAN »

## Conclusion

La gestion des boîtes de dialogue dans Sage X3 permet de créer une interface utilisateur dynamique et réactive. En utilisant différentes méthodes pour afficher des messages, poser des questions, et avertir les utilisateurs, les développeurs peuvent améliorer l'expérience utilisateur tout en fournissant des interactions nécessaires pour le bon déroulement des processus d'affaires. Les exemples fournis ici montrent comment intégrer ces techniques dans vos applications Sage X3 pour améliorer la communication avec l'utilisateur final.

# Configuration d'un fichier de log en utilisant des classes dans Sage X3

La création et la gestion d'un fichier de log sont cruciales pour le débogage et le suivi des opérations au sein d'applications Sage X3. L'utilisation de classes personnalisées pour gérer les fichiers de log permet une plus grande modularité et réutilisabilité du code. Cet article explore comment configurer et utiliser un fichier de log à travers une classe spécifique dans Sage X3.

## Définition de la Classe de Log

La classe de log, nommée C\_ALOG dans cet exemple, encapsule toutes les fonctionnalités nécessaires pour la manipulation des fichiers de log. Voici les étapes de base pour utiliser cette classe :

## Déclaration et Initialisation :

On commence par déclarer une variable locale pour stocker le nom du fichier de log :

« Local Char LOGFILE\_NAME(30) »

Ensuite, on déclare une instance de la classe C\_ALOG :

« Local Instance MYLOG Using C\_ALOG »

Création de l'Instance :

La mémoire est réservée pour l'instance de la classe :

« MYLOG = NewInstance C\_ALOG AllocGroup null »

Utilisation de la Classe :

On appelle la méthode pour initialiser le fichier de log :

« OK = fmet MYLOG.ABEGINLOG(" - TRAZA DE LOG - ") »

On récupère le nom du fichier de log automatiquement assigné :

« LOGFILE\_NAME = fmet MYLOG.AGETNAME »

Écriture dans le Fichier de Log

L'écriture des données dans le fichier de log se fait via la méthode APUTLINE de la classe C\_ALOG :

« OK = fmet MYLOG.APUTLINE("--------------", OK)

OK = fmet MYLOG.APUTLINE("LISTADO DE ARTICULOS", OK)

OK = fmet MYLOG.APUTLINE("--------------", OK)  »

Pour illustrer un cas d'utilisation, supposons que nous voulons logger tous les articles dont la référence commence par 'B' :

« Local File ITMMASTER [ITM] Where left$(ITMREF,1) = 'B'

Filter [ITM]

For [ITM]

OK = fmet MYLOG.APUTLINE([F:ITM]ITMREF, OK)

Next

Filter [ITM] »

Après avoir ajouté les informations nécessaires, on enregistre l'heure de fin de log :

« OK = fmet MYLOG.APUTLINE("--------------", OK)

OK = fmet MYLOG.APUTLINE(num$(time$), OK) »

Finalisation

Pour terminer l'enregistrement dans le fichier de log, on appelle la méthode de fin :

« OK = fmet MYLOG.AENDLOG »

Et on libère la mémoire assignée à l'instance de log :

« FreeGroup MYLOG »

Affichage du Nom de Fichier de Log

Enfin, on affiche le nom du fichier de log à l'utilisateur :

« Infbox LOGFILE\_NAME »

### Conclusion

L'utilisation d'une classe dédiée pour la gestion des fichiers de log dans Sage X3 simplifie la rédaction de code maintenable et réutilisable. Cette approche permet non seulement de centraliser la gestion des logs mais aussi d'ajuster facilement les fonctionnalités de logging sans modifier le code des processus métier.