Transactional Integrity

GeneXus™

GeneXus

Database Management System

Muitos gerenciadores de bases de dados (chamados DBMS) contam com sistemas de recuperação de falhas, que permitem deixar a base de dados em estado consistente quando ocorrem eventos imprevistos, como apagões ou quedas do sistema.

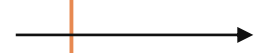
GeneXus

Logical Unit of Work (LUW)

...
Database Operation
Database Operation
LUW Ends

LUW Starts
Database Operation
Database Operation
Database Operation
Database Operation
LUW Ends

LUW



What if the system fails here?

Os gerenciadores de bases de dados (DBMSs) que oferecem integridade transacional permitem estabelecer unidades de trabalho lógicas (UTL), em Inglês
Logical Unit of Work, que correspondem nem mais nem menos que ao **conceito de "transações" de base de dados.**

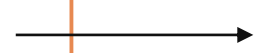
Basicamente, uma UTL é um conjunto de operações para a base de dados, as quais deverão ser executadas todas ou nenhuma. Em outras palavras, não é possível que dentro de uma UTL sejam executadas algumas operações e outras não. Isto nos garante a integridade da base de dados.

No exemplo, mostramos uma UTL que consiste em quatro operações sobre a base de dados, poderiam ser inserções, atualizações ou exclusões. Suponhamos que as duas primeiras tenham sido realizadas com sucesso, mas antes de executar a terceira, o sistema cai. Como a UTL não foi concluída, deverão ser desfeitas as duas operações que foram realizadas. Caso contrário, como as UTLs são as que definem em nível lógico os estados consistentes da base de dados, esta ficaria inconsistente.

Neste caso, o DBMS realiza o que é chamado de Rollback para se

recuperar, preservando o último estado consistente da base de dados.

Logical Unit of Work (LUW)

...
Database Operation
Database Operation
LUW Ends  Commit

LUW Starts
Database Operation
Database Operation
Database Operation
Database Operation
LUW Ends → Commit

LUW

What if the system fails here? Rollback

GeneXus

É o comando Commit o que determina o fim de uma UTL.
Portanto, uma UTL fica definida pelas operações realizadas entre dois commits.

Se o sistema cai onde se indica, as duas operações realizadas após o último Commit (que são as operações pendentes de Commit) são desfeitas com o Rollback automático que realiza o DBMS ao se recuperar da falha.

Ou seja, como não foram realizadas todas as operações que compõem a UTL, são desfeitas ou revertidas as operações parciais que foram realizadas.

GeneXus

LUW in GeneXus

Transactions: At the end of each instance, immediately before the AfterComplete rule.

Procedure: At the end of the Source

Business Component: GeneXus does not write Commit.

As transações e os procedimentos são os objetos GeneXus criados para atualizar a informação da base de dados. É por isso que GeneXus escreve o comando Commit ao gerar os programas na linguagem definida.

No objeto transação: ao final de cada instância, imediatamente antes das regras com evento de disparo AfterComplete (ou seja, após ter manipulado o cabeçalho e as linhas).
No objeto procedimento: ao final do Source.
Os Business Components, que são criados a partir das transações, não incluem Commit, pois podem ser utilizados em qualquer objeto, e será o **desenvolvedor quem determina onde "commitar".**
Veremos isso a seguir.

GeneXus

Transaction and Automatic Commit

Flight

SELECT

Id
Price
Discount Percentage
Airline Id
Airline Name
Airline Discount Percentage
Final Price
Capacity

Seat

Id Char Location

DELETE CANCEL CONFIRM

Header

BeforeValidate
VALIDATION
AfterValidate / BeforeInsert – BeforeUpdate - BeforeDelete
RECORDING
AfterInsert - AfterUpdate - AfterDelete

For each line

VALIDATION
AfterValidate / BeforeInsert – BeforeUpdate - BeforeDelete
RECORDING
AfterInsert - AfterUpdate - AfterDelete
END ITERATION LEVEL 2
AfterLevel Level attLevel2 / BeforeComplete
COMMIT
AfterComplete

O usuário manipula o cabeçalho e as linhas e pressiona “Confirm”. No servidor, são executadas as regras e fórmulas de acordo com a árvore de avaliação para o primeiro nível e, em seguida, são disparadas as regras condicionadas ao evento BeforeValidate. Depois que a informação do cabeçalho é considerada válida, são disparadas as regras condicionadas aos eventos AfterValidate e, dependendo do modo, as condicionadas a BeforeInsert, BeforeUpdate ou BeforeDelete. Depois disso, é gravado o cabeçalho e são disparadas as regras que estavam condicionadas a AfterInsert, AfterUpdate ou AfterDelete, conforme o modo.

Então, para cada linha:

São executadas as regras de acordo com a árvore de avaliação
São executadas as regras que tiveram evento de disparo BeforeValidate no nível das linhas.
É realizada a validação da linha (se a considera boa).
São executadas as regras que possuem evento de disparo AfterValidate ou, dependendo do modo, BeforeInsert, BeforeUpdate ou BeforeDelete.
É inserido/modificado/excluído o registro correspondente à linha na base de dados
São executadas as regras que possuem evento de disparo AfterInsert, AfterUpdate ou AfterDelete, dependendo do modo da linha.

Ao finalizar com a última linha, são executadas as regras que possuem como evento de disparo After Level de um atributo do segundo nível.
Se existe outro nível paralelo, é repetido o mesmo para esse outro nível.
Quando é concluído o último nível, são disparadas as regras que estejam condicionadas ao evento BeforeComplete. Depois disto é que GeneXus coloca o comando Commit automaticamente. Portanto, é executado o Commit. Ou seja, ficam commitadas as informações do cabeçalho e linhas. Em seguida, são disparadas as regras que foram condicionadas ao evento AfterComplete.

GeneXus

Transaction and Automatic Commit

Invoice 1	Invoice 2	Invoice 3
Record Header 1	Record Header 2	Record Header 3
Record Line 1.1	Record Line 2.1	Record Line 3.1
Record Line 1.2	Record Line 2.2	Record Line 3.2
Commit	**Commit**	Record Line 3.3
		Record Line 3.4
		Record Line 3.5

LUW **LUW**

Transaction integrity	
Commit on exit	Yes

Suponhamos que o usuário deseja inserir 3 faturas no sistema. Insere a primeira, insere a segunda, e quando Confirma a terceira, imaginemos que quando o programa está processando a terceira linha, após gravá-la, o sistema falha e a base de dados deve ser iniciada novamente. Em que estado estará a base de dados?

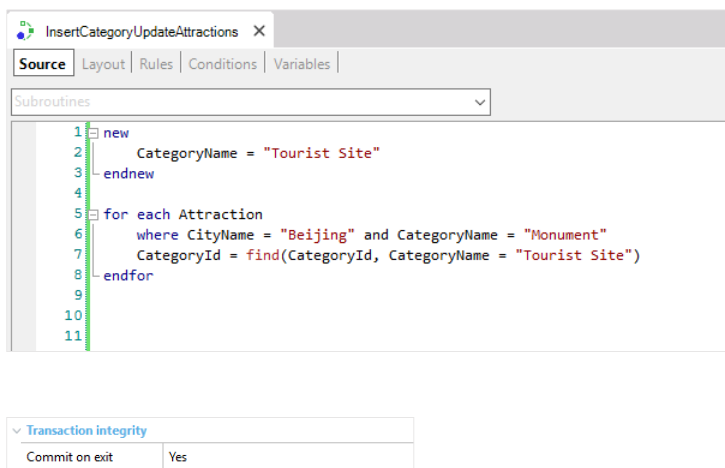
Como o DBMS fará um rollback, desfará todas as operações que não foram "commitadas". No nosso caso, serão eliminados os registros correspondentes ao cabeçalho e às três linhas da fatura 3.

Observe que se tivesse sido desabilitado o commit automático da transação Invoice (propriedade "Commit on exit" = "No") nenhum dos registros inseridos (nem os da invoice 1 nem os da 2 e, claro, nem os da 3) permanecerá na base de dados. Neste caso, todas as operações formarão uma única UTL, enquanto se for deixado o valor padrão para a propriedade Commit on Exit, "Yes", cada cabeçalho com suas linhas formarão uma UTL diferente.

Para transações, é recomendado configurar esta propriedade como Yes.

GeneXus

Procedure and Automatic Commit



Em todo procedimento que acessa a base de dados, GeneXus adicionará automaticamente (salvo seja indicado o contrário através da propriedade Commit on exit) um Commit.

Como os procedimentos são usados para outras coisas além de atualizar a base de dados (por exemplo, para listar informações ou realizar cálculos ou simplesmente consultar a base de dados) GeneXus inserirá automaticamente o comando Commit no programa gerado se entender que esse procedimento tenta atualizar a base de dados. Caso contrário, não o coloca, independentemente do valor da propriedade Commit on Exit.

Neste caso, em que estão sendo usados os comandos new para inserir uma categoria e o for each para atualizar o atributo CategoryId da tabela associada à transação Attraction, dado que a propriedade Commit on exit está em Yes, GeneXus escreverá automaticamente o Commit no programa gerado, ao final do código. Isto faz com que, se após ter inserido a nova categoria na tabela CATEGORY, e ao modificar o terceiro monumento de Beijing, mudando a categoria para a nova, se o sistema cai, nenhuma das mudanças anteriores (nem a categoria nova, nem as mudanças nos dois monumentos anteriores) permanecerá na base de dados. Todas as operações deste procedimento serão parte de uma

mesma UTL. Onde começa essa UTL?

Dependerá de quando foi realizado o último Commit. Se após a última operação na base de dados realizada antes de invocar este procedimento foi realizado um Commit, então a UTL inicia com o new deste procedimento. Caso contrário, todo este código fará parte de uma UTL que começou antes. Onde? Onde se encontre o Commit anterior, imediatamente depois.

Onde termina a UTL? Se a propriedade Commit on Exit está em "Yes", termina ao final do procedimento. Senão, termina onde se encontre o próximo Commit (teremos que ver no que chamou este procedimento, que é o que segue sua invocação).

GeneXus

Procedure and Explicit Commit

Explicit Commit

Source * | Layout | Rules | Conditions | Variables

Subroutines

```
1 &Category.CategoryName = "Tourist site"
2 &Category.Save()
3
4 If &Category.Success()
5     Commit
6 endif
7
```

Automatic Commit

Source * | Layout | Rules | Conditions | Variables

Subroutines

```
1 new
2     CategoryName = "Tourist Site"
3 endnew
4
5 for each Attraction
6     where CityName = "Beijing" and CategoryName = "Monument"
7     CategoryId = find(CategoryId, CategoryName = "Tourist Site")
8 endfor
9
```

New

For each that updates

Delete

GeneXus reconhece que deve realizar um acesso à base de dados dentro de um procedimento quando se utiliza o comando new, for each para atualizar ou comando delete. Nestes casos coloca o Commit implícito. Caso contrário, não entende que há acesso à base de dados, e é por isso que quando em um procedimento é feito o seguinte. Ou mesmo se em vez de .Save() fazemos .Insert(), .Update() ou .InsertOrUpdate(). Não adiciona o Commit. GeneXus não percebe que estamos querendo fazer um Insert na base de dados, mesmo utilizando o método insert, portanto deve ser escrito o comando Commit explicitamente.

Como acabamos de dizer, se dentro do código do procedimento existe um comando new, um for each que atualiza ou um comando Delete, em qualquer um desses casos, não teria que escrever explicitamente o Commit. Se só temos em nosso procedimento este código, ali sim teremos que adicioná-lo de forma explícita.

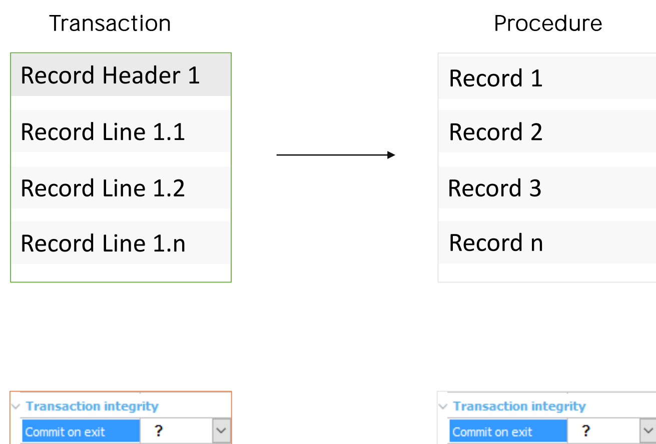
GeneXus

Customizing LUWs

No exemplo que vimos, Genexus colocava um commit no final automaticamente. Mas se quiséssemos que a gravação da categoria seja parte de uma UTL e as gravações das atrações sejam parte de outra, de forma que se o sistema cai antes de terminar de modificar as atrações, a categoria fique inserida, mas as atrações não, como fazemos?

Será através do uso do Commit. Teremos que escrever um Commit após ter inserido a categoria, e outro após ter inserido todas as atrações. Este segundo, podemos ignorá-lo se tivermos ativada a propriedade Commit on exit. De qualquer forma, parece uma boa prática escrevê-lo explicitamente, caso posteriormente sejam adicionadas mais operações ao Source do procedimento, que devam ficar em outra UTL.

Customizing LUWs

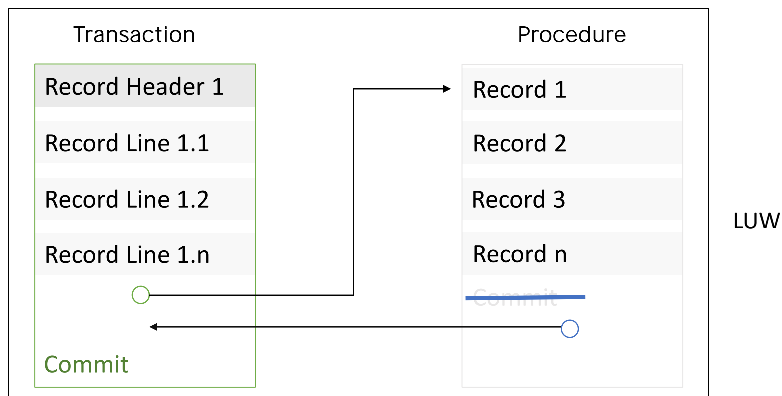


Se precisamos invocar a partir de uma transação "A" a um procedimento "B" que realiza operações na base de dados, de forma que as atualizações do registro do cabeçalho da transação e todas as linhas, bem como de todos os registros do procedimento formem uma única UTL (e, portanto, se o sistema cai antes de concluir tudo, sejam desfeitas todas as alterações), o que devemos programar?

Temos várias possibilidades para conseguir isso.

GeneXus

Customizing LUWs



Procedure (parm1, parmN) on BeforeComplete;

Transaction integrity — Commit on exit: Yes

Transaction integrity — Commit on exit: No

Uma alternativa é realizar as seguintes etapas:

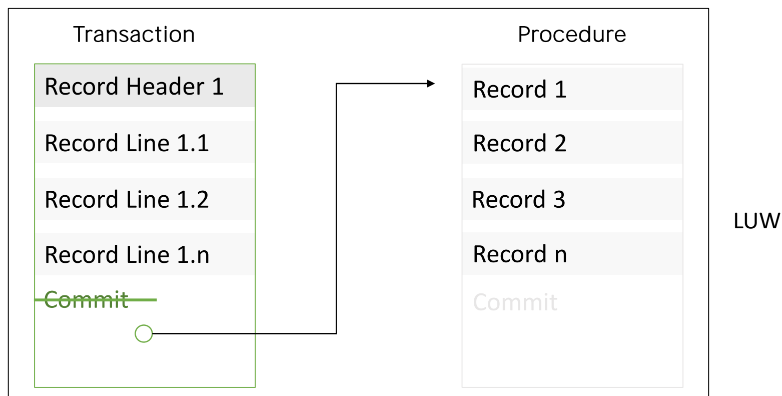
1 Invocar o procedimento em algum momento ANTERIOR ao Commit **automático. Por exemplo "on BeforeComplete"**

2. Desabilitar o Commit automático do procedimento.

Desta forma, estará sendo invocado o procedimento após terem sido gravados todos os registros (o correspondente ao cabeçalho e os correspondentes às linhas), já tendo disparado todas as regras condicionadas ao evento AfterLevel. Ou seja, estará sendo chamado o procedimento um instante antes do Commit. O procedimento realizará todas as suas atualizações de registros da base de dados e, como desabilitamos seu Commit, se o desenvolvedor não incorporou este comando explicitamente dentro de seu código, a UTL não será fechada. Depois de finalizada a execução do código do procedimento, retorna-se ao chamador, para a sentença que segue a invocação. Aqui é onde se encontrará o Commit.

GeneXus

Customizing LUWs



Procedure (parm1, parmN) on AfterComplete;

Transaction integrity	
Commit on exit	No

Transaction integrity	
Commit on exit	Yes

Outra alternativa é realizar as seguintes etapas:

1. Não importa quando se invoca o procedimento, desde que seja depois de ter gravado todos os registros (do cabeçalho e das linhas) da transação, mesmo depois de onde iria o Commit: Por exemplo, on AfterComplete.

2. Contanto que seja desabilitado esse Commit automático da transação e seja deixado o commit automático do procedimento.

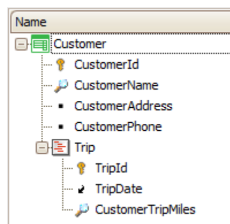
Invocando o procedimento neste momento, estamos seguros de que todos os registros do cabeçalho e linhas terão sido gravados. Então o procedimento realizará suas atualizações de registros na base de dados, e fará seu commit, commitando, portanto, todos os registros (o seus e os da transação).

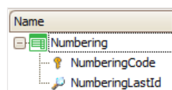
Essas são apenas duas das múltiplas alternativas. A escolhida dependerá da lógica que está querendo implementar (normalmente não será indiferente o momento de invocação ao procedimento).

Customizing LUWs

GeneXus

Transactions






```
CustomerId = GetNextNumber( "Customer" )
    on BeforeInsert;
```

Transaction integrity

Commit on exit: Yes

Procedure

GetNextNumber *

Source * | Layout | Rules | Conditions | Variables

Subroutines

```
1  for each Numbering
2      where NumberingCode = &who
3      &nextNumber = NumberingLastId +1
4      NumberingLastId = &nextNumber
5  when none
6      new
7          NumberingCode = &who
8          &nextNumber = 1
9          NumberingLastId = &nextNumber
10     endnew
11 endfor
12
13
```

```
parm( in: &who, out: &nextNumber );
```

Transaction integrity

Commit on exit: No

Dada a transação Customer que registra os clientes e suas excursões contratadas.

Suponhamos que não queremos usar a estratégia de numeração automática da base de dados, mas queremos ter uma tabela interna própria na base de dados que registre o último número dado a cada entidade para numerar seu identificador.

Para fazer isso, criamos uma transação chamada Numbering cujo atributo identificador NumberingCode registra o nome da entidade em questão (por exemplo "Customer", "Trip", "Invoice", "Category", etc.) e seu atributo NumberingLastId, o último número fornecido para essa entidade.

Em seguida, programaremos um procedimento, GetNextNumber, que será o responsável por obter o próximo número a ser atribuído ao identificador da entidade que o está chamando.

Analisaremos o código em um momento.

Por exemplo, a partir da transação Customer, quando o usuário deseja inserir um novo cliente, deixará vazio o campo CustomerId na tela e ao sair do campo, passando ao próximo campo, CustomerName, a transação

saberá que está querendo inserir um registro novo (ficará em modo "INS", insert).
Como não temos o atributo CustomerId autonumerado, teremos que invocar o procedimento GetNextNumber, para obter o número a ser atribuído ao CustomerId.
Deveremos passar por parâmetro para o procedimento quem somos, ou seja, neste caso, "Customer" para que o procedimento busque na tabela Numbering o último número que foi dado a um "customer".

Vejamos em detalhes o que faz o procedimento criado.
Em primeiro lugar, nas regras declaramos dois parâmetros, um de entrada e outro de saída. O parâmetro de entrada será aquele que utilizaremos para saber o que queremos numerar, um cliente, uma viagem, uma fatura, etc. Esta variável será do tipo character.
E de saída declaramos a variável NextNumber, do tipo numérico, que devolverá o valor que nos interessa.

Voltando ao source, será percorrida a tabela da transação Numbering.
Com o where indicamos que queremos recuperar o registro em que NumberingCode tenha o mesmo valor da variável que passamos por parâmetro, neste caso Customers.
Se o encontra, ao atributo NumberingLastId será adicionado um, e esse valor será atribuído à variável nextNumber. Em seguida, ao atributo NumberingLastId atribuímos o valor de nextNumber.

Caso não seja encontrado um registro em que NumberingCode tenha o valor da variável que lhe passamos, criaremos um registro na base de dados utilizando o comando new. Em que NumberingCode terá o valor da variável passada por parâmetro, por exemplo Invoice. Em seguida, atribuímos à variável nextNumber o valor um, e ao atributo NumberingLastId atribuímos o valor de nextNumber, já que sendo um novo registro nesta tabela iniciaremos com este valor.

Se invocarmos este procedimento por meio da regra de atribuição sem evento de disparo, por exemplo if Insert:
CustomerId = GetNextNumber("Customer") if Insert; o procedimento será disparado:

Uma vez, assim que o usuário na tela deixe vazio o campo CustomerId e saia do campo.
Uma segunda vez, quando o usuário confirme, e as regras novamente são disparadas em ordem, no servidor.

Imaginemos que o último id de cliente é o número 5.

O procedimento será executado atualizando para 6 o registro da tabela Numbering correspondente e mostrando ao usuário imediatamente na tela o número 6. Mas o que acontece se o usuário se arrepender e nunca pressionar Confirm, mas cancela? O número 6 terá sido perdido.

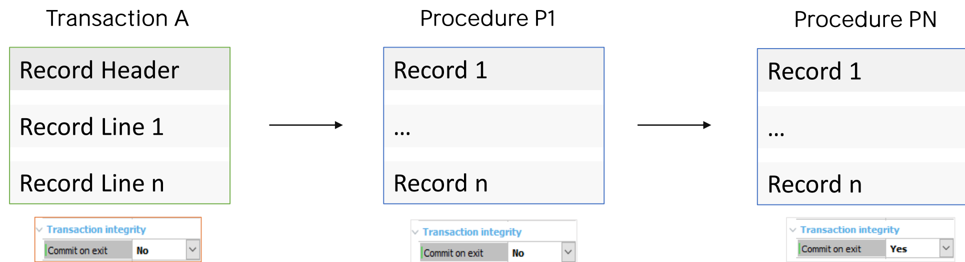
Mesmo que o usuário confirme, novamente será executado o procedimento, e o número que será atribuído ao cliente será o número 7, portanto o número 6 também será perdido.

Como resolvemos esta situação? Condicionando a invocação do procedimento a um evento de disparo (desta forma a regra de atribuição não será disparada no cliente, enquanto o usuário trabalha na tela). Qual é o momento apropriado? O último momento possível é BeforeInsert do cabeçalho. Por quê? Pois a partir desse momento, o valor que atribuímos a qualquer um de seus atributos não terá efeito, pois o registro já terá sido gravado.

Agora, o procedimento GetNextNumber modifica um registro da tabela Numbering, ou insere um se não existia. Portanto, por padrão, colocará um Commit automático no final. Desta forma, se imediatamente após retornar à transação, suponhamos que se insere o cliente em sua tabela, e ao inserir a terceira excursão o sistema cai, ao ser restabelecido, não ficarão registrados os registros associados ao cliente, mas será perdido esse número, uma vez que já havia sido commitado. Para que todas as operações realizadas por transação e procedimento fiquem dentro da mesma UTL, conseguirá neste caso, com não fazer Commit on Exit no procedimento, e que o Commit de tudo seja feito pela transação.

GeneXus

Customizing LUWs

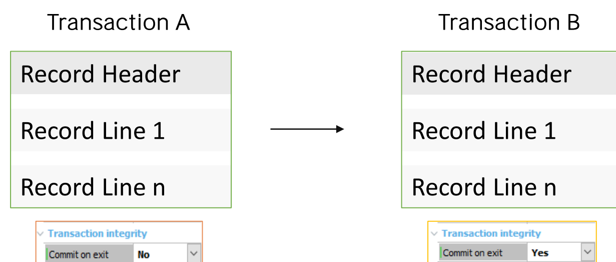


No puede conformarse una única UTL entre transacciones.

Si desde una transacción se invoca a un procedimiento que invoca a otro procedimiento, todos con el Commit deshabilitado salvo el último (o también puede tener esté su commit deshabilitado y quien hace Commit es la transacción cuando se le retorna el control) su commit commiteará los registros de la transacción y de todos los procs.

GeneXus

Customizing LUWs

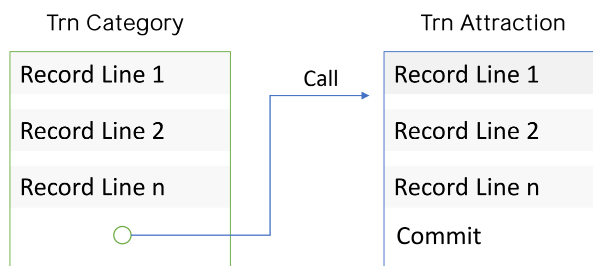


No entanto, se a transação ("A") chama outra transação ("B"), o commit da segunda ("B") NÃO commitará os registros manipulados pela primeira ("A"). São Commits independentes. Então se desabilitamos o commit de "A" e chamamos "B", que faz Commit, os registros de "A" não serão commitados por ninguém!

Como fazemos então para conseguir que sejam inseridos o cabeçalho e linhas de uma transação "A", seja chamada uma "B" para inserir informações de certa forma relacionadas, e que todas estas operações formem uma única UTL?

GeneXus

Customizing LUWs



A partir do trabalhar com Categories, queremos que, ao pressionar Insert, seja oferecida ao usuário a possibilidade de inserir uma nova categoria e imediatamente uma atração dessa categoria. Porque não queremos deixar inseridas categorias sem atrações relacionadas.

Para isso, a partir da transação Category, bastará invocar a transação Attraction on AfterInsert (para que o CategoryId já tenha o valor autonumerado correto, fornecido pela base de dados), e declarar em Attraction que receberá um parâmetro.

Desta forma, quando inserimos uma Categoria, nos levará à transação Attraction para inserir uma atração. Já ficando selecionada a categoria que acabamos de passar por parâmetro.

Mas... o que acontece se o sistema cai imediatamente depois que a Attraction tenha feito seu commit? Esse Commit, terá commitado também o registro de Category que já havia sido inserido? A resposta é Não, pelo que vimos antes.

Necessitamos que a inserção de uma categoria e depois, de uma atração, formem uma mesma UTL e o Commit final commite ambos os registros. Como conseguimos isso?

Veremos duas opções, das muitas que existem.

GeneXus

Customizing LUWs

WorkWithCategoy

```
Event Start
    If not IsAuthorized(&PgmName)
        NotAuthorized(&PgmName)
    Endif

    Grid.Rows = 10
    &Update = "GXM_update"
    &Delete = "GX_BtnDelete"
    Form.Caption = 'Categories'

    Do 'PrepareTransaction'
EndEvent

Event Grid.Load
    &Update.Link = Category.Link(TrnMode.Update, CategoryId)
    &Delete.Link = Category.Link(TrnMode.Delete, CategoryId)
    CategoryName.Link = ViewCategory.Link(CategoryId, "")
EndEvent

Event 'DoInsert'
//  Category(TrnMode.Insert, nullvalue(CategoryId))
InsertCatagoryAndAttraction()
EndEvent
```

InsertCategoryAndAttraction

<ErrorViewer: ErrorViewer1>

Category

Category Name &Category.CategoryName

Attraction

Attraction Name &Attraction.AttractionName

Confirm Cancel

```
Event 'Confirm'
    If not (&Category.Insert())
        for &message in &Category.GetMessages()
            Msg(&message.Description)
        endfor
    else
        &Attraction.CategoryId = &Category.CategoryId
        &Attraction.Save()
        for &message in &Attraction.GetMessages()
            Msg(&message.Description)
        endfor
        if &Attraction.Success()
            Commit
            return
        else
            Rollback
        endif
    endif
Endevent
```

Uma solução será que o botão Insert do “trabalhar com” invoque um web panel, que é um objeto com interface onde os desenvolvedores programam livremente o comportamento.

Nele, inserimos duas variáveis &category e &attraction em seu form. Serão Business Components de Category e Attraction, respectivamente.
Só nos interessa que o usuário insira o nome da categoria que está querendo criar e o nome da atração dessa categoria. Ao fazer isso e pressionar Confirm será disparado o evento associado, que programamos conforme visto na tela de eventos.

Vamos levar em consideração, para este exemplo, que os atributos CountryId e AttractionId, das transações Country e Attraction respectivamente, têm a propriedade autonumber em true.

Primeiro tentamos fazer o Insert do BC de Category. Se falhar (por exemplo, se o usuário deixou vazio o nome da categoria na tela e a transação possui uma regra error para evitar esse caso) mostramos no controle ErrorViewer as mensagens produzidas pelo BC.
Nesse exemplo, a mensagem de erro é disparada duas vezes, uma do lado do cliente quando deixamos o campo vazio. E outra quando confirmamos, do lado do servidor. Ambos os casos dão o mesmo resultado. Chamamos

estes tipos de regras de idempotentes.

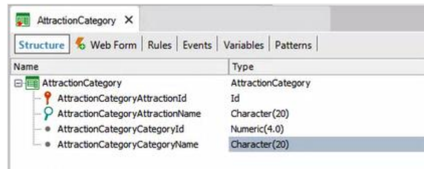
Se inserimos todos os dados corretamente, então à variável BC de Attraction atribuímos como categoria a recém-inserida e tentamos fazer o Save. Mostramos as mensagens produzidas e depois se a operação foi bem-sucedida, fazemos Commit, após o que, agora sim, ficarão commitados ambos os registros.

Se a operação falhou, então observemos que contamos com o comando Rollback, para desfazer a inserção do registro de Category que havia sido inserido com sucesso.

GeneXus

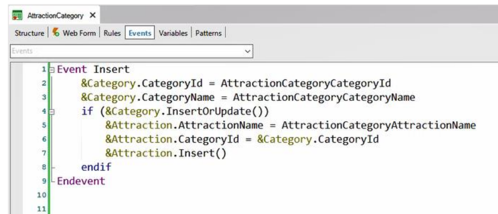
Customizing LUWs

Trn AttractionCategory



Data Provider

Insert Event



Uma segunda alternativa para resolver o mesmo requisito é utilizar uma transação dinâmica em vez do web panel.

Lembremos que as transações dinâmicas são aquelas que não geram tabela física, são obtidos os dados consultados em tempo de execução. São definidas configurando as seguintes propriedades da transação. Data Provider em true, e Used To em Retrieve Data.
Ao definir o Data Provider como true, será criado automaticamente um objeto Data Provider. E ao configurar Used To em Retrieve Data, GeneXus entenderá que não deverá criar a tabela associada à transação, já que nesse Data Provider se declarará de onde obter os dados.

No exemplo criamos a transação dinâmica AttractionCategory, cujas informações serão retiradas da tabela de atrações, sabendo que cada atração possui uma categoria. Será como uma visão de atrações.

Quando o usuário queira inserir nesta transação uma atração, terá permissão para inserir os dados da atração, bem como os da categoria. No evento Insert utilizamos a variável &category business component de Category, e a variável &attraction business component de Attraction e copiamos para seus membros os valores que o usuário especificou nos atributos da transação dinâmica, através do seu form.

Se a categoria não existe, é criada antes de inserir a atração. Se existe, é permitido atualizar seu nome. Em seguida, é inserida a atração com a categoria.
Se deixamos a propriedade Commit on Exit da transação com seu valor padrão, Yes, após a execução do evento Insert, por se tratar de uma transação de um só nível, será realizado o Commit, que terá efeito sobre os dois registros inseridos através dos business components.

GeneXus™

training.genexus.com
wiki.genexus.com