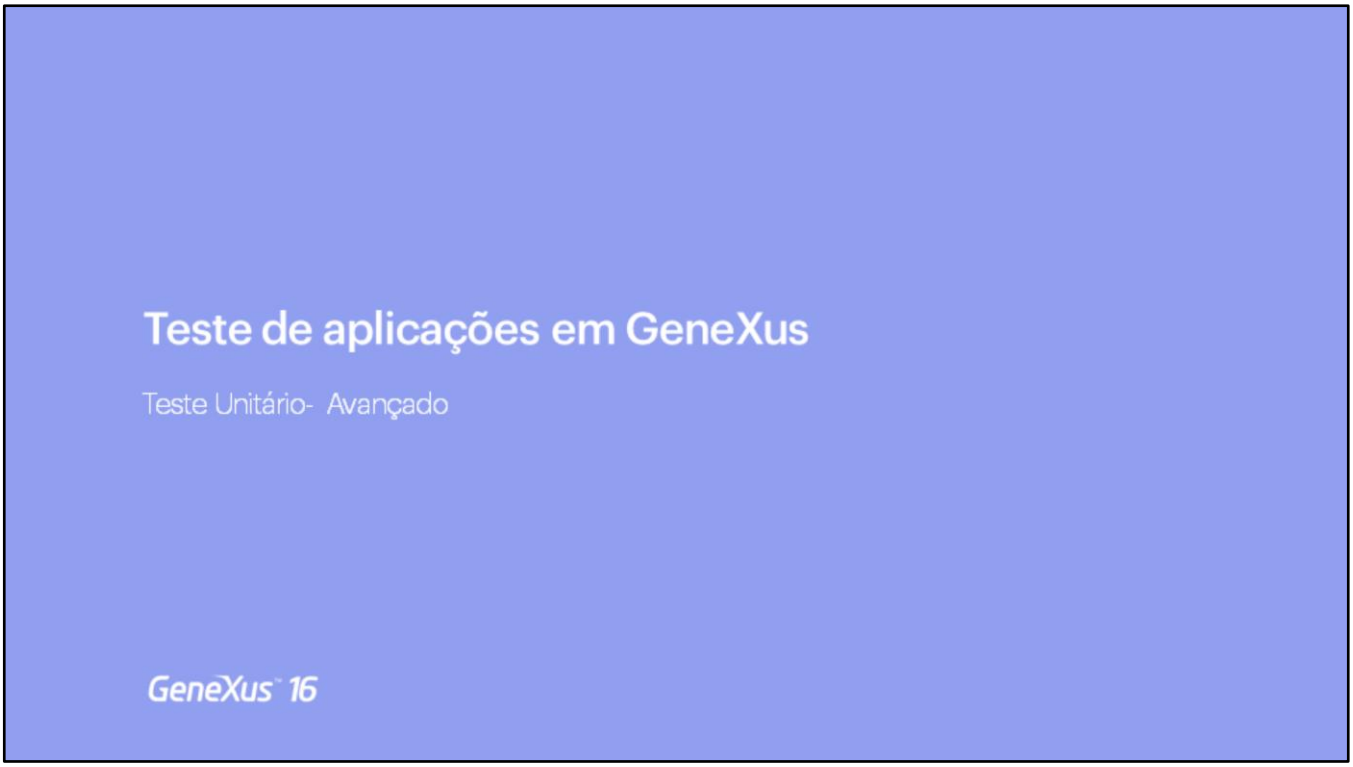
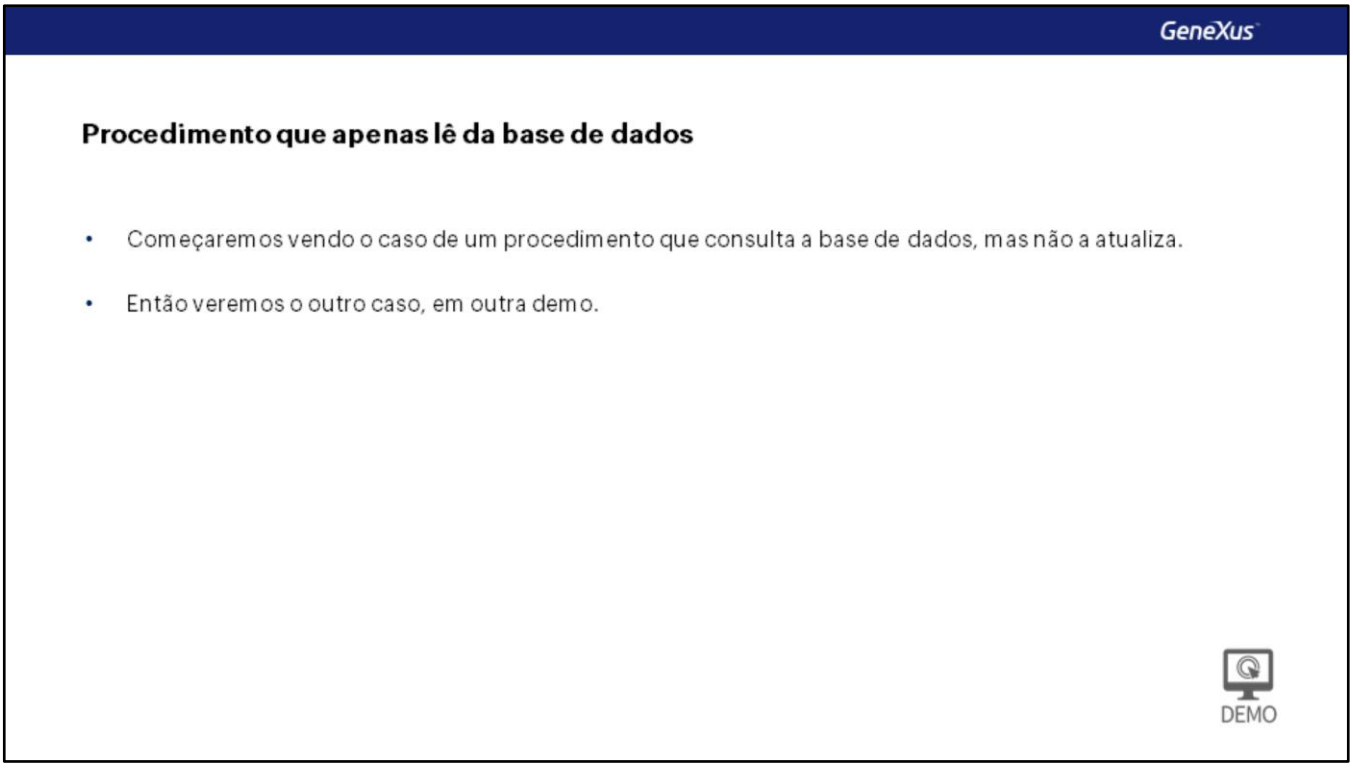
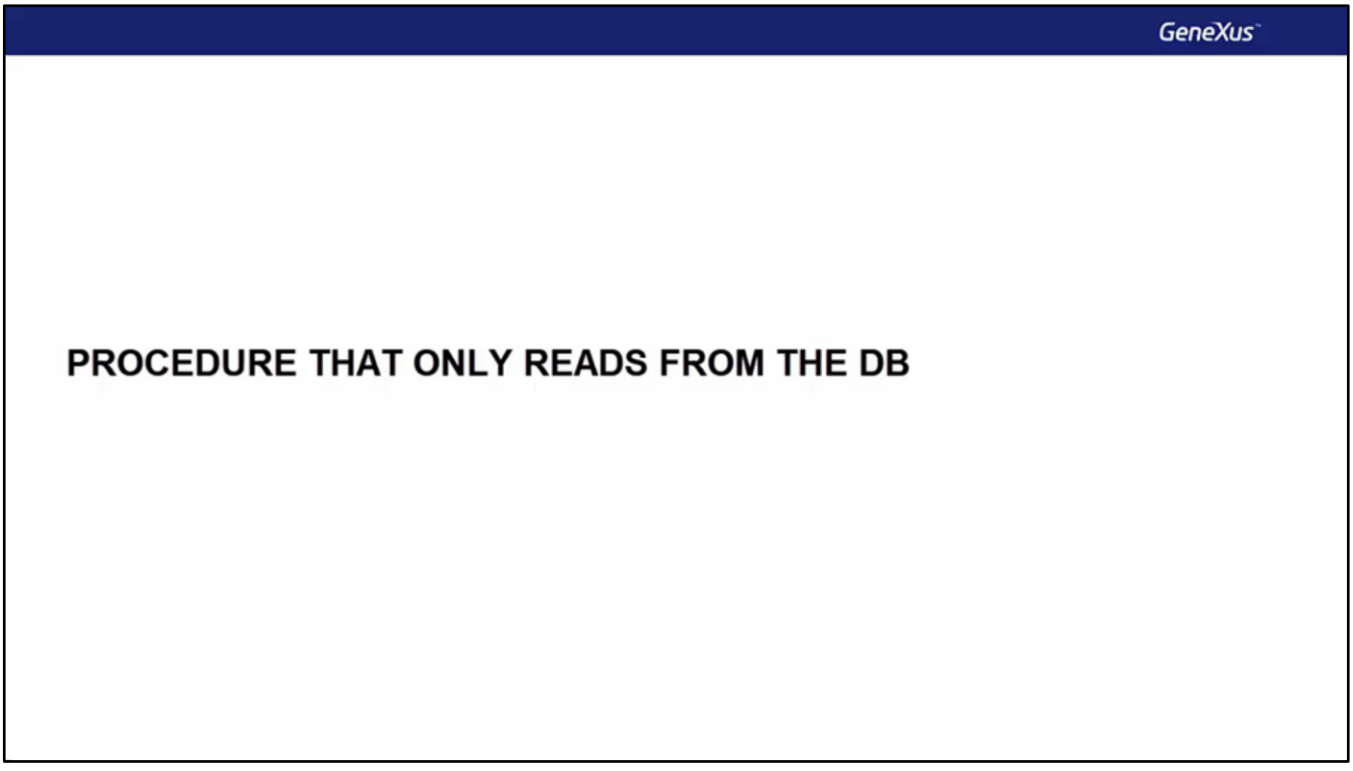


Agora que vimos um exemplo básico de como fazer um teste unitário no GeneXus, veremos alguns exemplos mais avançados interagindo com a base de dados.



Começaremos vendo um procedimento que consulta a base de dados, mas não a atualiza.



[DEMO: https://youtu.be/PJjtTEUTaO8]

Temos um procedimento "doLoadCountries", que é um serviço que utilizaremos para alimentar os diferentes relatórios de países na aplicação. O serviço recebe como parâmetro de entrada um SDT que chamamos CountryREportOptions, este SDT tem um elemento onde definimos os filtros pelos quais vamos selecionar os países. Por enquanto, só fazemos isso com base no número mínimo de atrações que queremos que o país tenha.
E então tem outro elemento SortCriteria onde podemos definir um critério de Ordem
Este é um enumerado que nos permite retornar a lista de países ordenada por nome ou por quantidade de atrações.

Em seguida, este serviço recebe um critério de filtro e um de ordem e retorna uma variável chamada SDTCountryCollection, onde está a coleção de países que obedece ao filtro especificado é está classificada pelo filtro especificado.

O benefício de implementar este serviço é que podemos realizar testes unitários automáticos, que validam que os dados dos relatórios estão corretos em todos os cenários que consideramos importantes testar, então programamos os relatórios usando este serviço como fonte de dados em vez de acessar diretamente a BD, desta forma, quando vamos testar os relatórios, já temos confiança de que os dados relatados estão corretos, graças ao teste de unitário que fizemos previamente e só precisamos verificar se o formato do relatório está correto.

Este exemplo é mais complexo do que o procedimento de Discount porque os parâmetros de entrada e saída são SDTS em vez de dados simples e, portanto, a carga dos casos de teste nos dará um pouco mais de trabalho e também porque neste caso estaremos acessando a base de dados, por isso ao escolher os valores para testar, precisamos conhecer a base de dados que estamos testando

Vamos criar o teste unitário e continuaremos falando sobre estes conceitos à medida que avançamos

Algo importante a destacar é que os objetos que são criados para o teste unitário ficam aninhados sob o objeto a ser testado e não são gerados quando fazemos um build-all da KB. São gerados somente quando é necessário para executar os testes automáticos.

O fato de ficarem aninhados sob o procedimento significa simplesmente que, se excluirmos o procedimento, os testes associados serão excluídos, os objetos não estão sincronizados no sentido de que, se modificarmos os parâmetros de entrada ou os parâmetros de saída do procedimento doloadcountries teremos que atualizar manualmente também os testes e, de fato, teremos que pensar sobre o impacto das mudanças nos parâmetros de entrada e saída em nossos casos de teste.

E falando em casos de teste, para executar nosso teste unitário, a primeira coisa é definir quais são os casos de teste – ou pelo menos 1 caso de teste com o qual queremos testar.
Vejamos o dataprovider que foi gerado automaticamente e vemos que é um pouco diferente do caso do Discount onde todos os parâmetros de entrada e saída eram simples, neste caso como são SDTs o dataprovider gera automaticamente uma estrutura que nós devemos completar

Para isso, podemos clicar e arrastar os SDTs de forma a trazer a estrutura do SDT para o dataprovider como sempre fazemos, ou alternativamente, neste caso como é simples, escreveremos.

Em nosso primeiro caso de teste, testaremos um caso que serve para o relatório que mostre aqueles países com mais de 2 atrações e os retorna mostrando os países com maior quantidade de atrações acima. Para definir os valores esperados, temos 2 opções.. Uma é ir à DB, analisar os valores e entrar e inseri-los aqui nos resultados do caso de teste. Alternativamente, o que faremos agora é executar o teste com um resultado esperado Vazio – que sabemos que irá falhar – veremos qual é o resultado obtido, vamos validá-lo à mão e, caso estejamos satisfeitos com esses valores, copiaremos esses valores como o resultado esperado.

A vantagem disto é que podemos tirar uma foto do resultado do procedimento – dados certos valores de entrada – e se no futuro o procedimento for modificado, teremos uma maneira muito fácil de saber se nada foi danificado, isto é, se em comparação com os mesmos dados de entrada, são geradas a mesma saída ou os mesmos dados do relatório.

É muito importante conhecer a bd em que estamos testando, inclusive ter uma BD estável para executar os testes, de forma a não ter testes que falhem devido a mudanças na BD.

Como esperado, o teste falha, agora não podemos ver na tela Test-Results o valor esperado e o valor obtido porque o resultado do nosso procedimento é um SDT – que aqui está sendo manipulado como um json para poder compará-lo como string, para poder vê-lo em detalhes, temos a opção de expandir o resultado e vê-lo em um comparador de texto

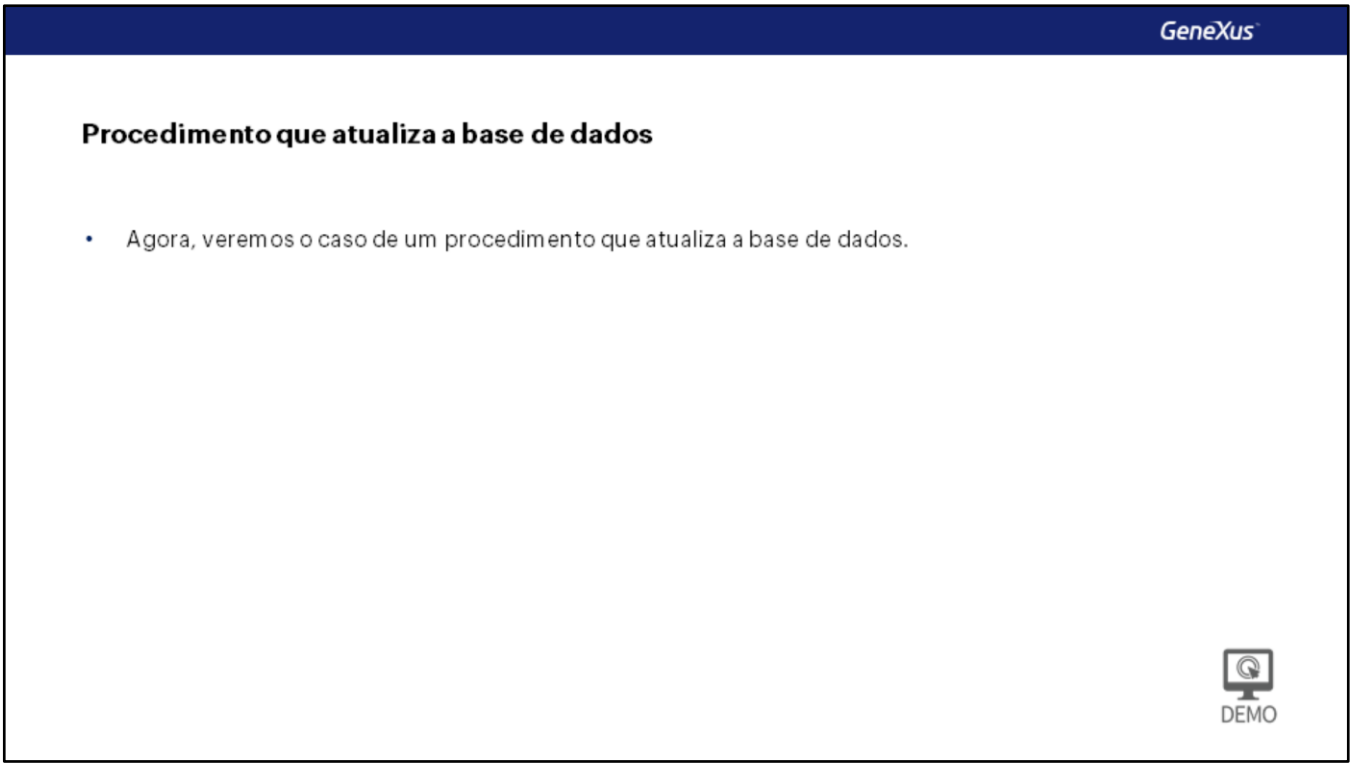
No nosso caso, vemos que o resultado obtido é que França e China são os países que possuem 2 ou mais atrações, o valor esperado está em Branco porque não o definimos. Se verificarmos que estes são os valores corretos, podemos agora colocar esta informação como resultado esperado em nosso data Provider.

Aqui podemos copiar e colar a informação para o dataprovider, embora o formato do resultado seja um json, então teremos que dar-lhe a estrutura correta em nosso data-Provider

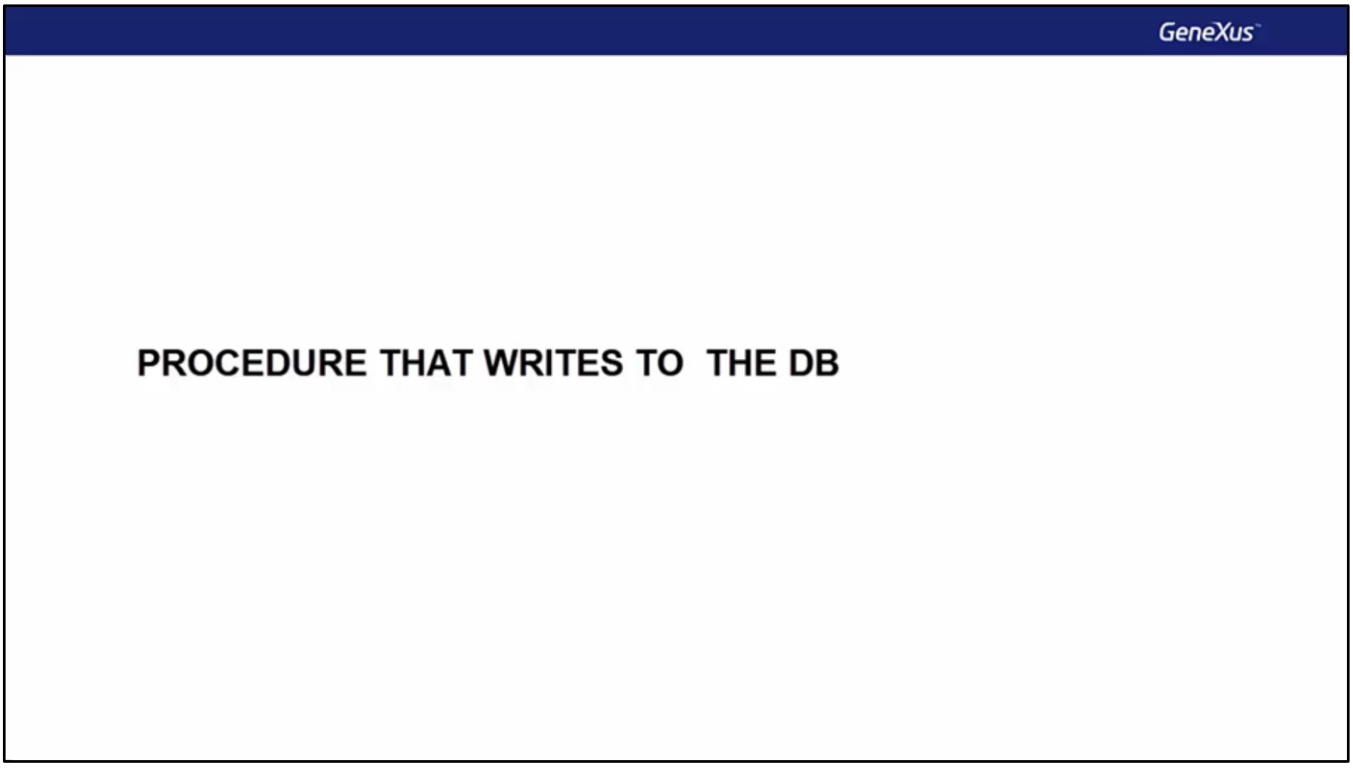
Voltemos a executar e agora sabemos que teremos um teste bem-sucedido

Podemos então continuar adicionando casos de teste, neste caso o procedimento é muito simples, então poderíamos adicionar um caso que provaria outro critério de ordem e talvez um filtro vazio.. Em um caso mais complexo, que tivesse mais critérios de filtro certamente deveríamos ter um conjunto de testes mais rico.

Não veremos neste curso, mas existe uma maneira de pular a BD para que, enquanto nada mudar no nível do procedimento testado, seja executado contra um estado salvo da BD, mas não contra a BD real. Isso é chamado de mocking de dados e você pode obter mais informações em nosso wiki



Vejamos um segundo exemplo, agora testando um procedimento que faz mudanças na BD.



[DEMO: https://youtu.be/XUkq5T3qRV4]
Para isto, veremos AttractionInsert, que é um procedimento que insere atrações em nossa BD.
Este procedimento recebe como parâmetros uma estrutura de dados com as informações da atração, realiza o insert e então retorna como resultado uma estrutura de dados que possui um código de resultado e uma mensagem, além do ID da atração criada

Vamos observar a implementação deste procedimento

Algo importante a notar é que o procedimento utiliza um BC para fazer a inserção, portanto, quando testamos este procedimento, não estamos apenas testando o procedimento em si, mas também as regras que definimos no BC.

Para criar o teste, fazemos o mesmo de sempre, clicamos o botão direito e selecionamos “Create Unit Test”

Então vamos definir como sempre os casos de teste. O parâmetro de entrada do nosso procedimento era um AttracionSdt, então como não nos lembramos da estrutura dele vamos procurá-lo na KB e clicamos nele e arrastamos para trazer a estrutura aqui. Então completamos os valores de acordo com o caso que queremos testar. Nosso primeiro caso de teste será o caso feliz, ou seja, vamos colocar valores válidos para que uma nova atração possa ser registrada com sucesso

Faremos o mesmo para o resultado esperado, que neste caso é um InsertResponse, então procuraremos pela estrutura InsertResponse, a traremos para o dataprovider e definiremos aqui o que esperamos. A estrutura de InsertRESPONSE tem um código de resultado que no nosso caso esperamos que seja bem-sucedido, uma mensagem que no caso onde a inserção seja bem-sucedida é vazia, e o ID da atração que será inserida.

Note que aqui deixamos em branco porque o ID a ser atribuído é autonumerado e não podemos saber a priori qual será o valor atribuído. Se executarmos este teste e atribuirmos um valor aqui no DataProvider, o teste poderia funcionar apenas na primeira vez - se soubéssemos qual é o próximo autonumerado a ser atribuído -, mas a segunda vez que quiséssemos executar o autonumerado seria diferente e o teste falharia, então para evitar este problema o que teremos que fazer é modificar o teste unitário

Para isso, o que faremos é, uma vez que tenha sido invocado o procedimento AttractionInsert e já tenhamos obtido uma resposta, modificaremos nosso valor esperado com o valor realmente obtido, desta forma, estes valores serão sempre iguais. Ou seja, o valor esperado e o valor obtido do ID sempre virão do mesmo local e, portanto, o Assert apenas desse elemento será um passe, os que nos interessam são os outros, ou seja, o código de resultado e a mensagem

Vamos executar nosso teste e, enquanto executa, uma coisa importante a observar é que a única coisa que estamos validando neste teste, ou seja, o único sobre o qual estamos fazendo Assert é o código de resultado e a mensagem que retorna o procedimento, mas não estamos validando realmente que esteja ficando a informação bem registrada na base de dados.

Para fazer isso, temos que modificar nosso teste unitário, para adicionar algum Assert que esteja verificando os dados que esperamos. Agora podemos executar nosso teste novamente e agora sim sabemos que validaremos os dados na base.

Note que nosso teste falhou, e é que existe uma chave única para o nome e o país da atração, então precisamos introduzir algum tipo de variação no nome da atração para executar este teste quantas vezes quisermos, para isso vamos apenas atribuir um time-stamp.

Agora podemos voltar a executar o nosso teste e desta vez será bem-sucedido
Vemos que temos vários Assert, ou vários resultados, um para cada Assert que está sendo feito no teste unitário

Agora que temos nosso primeiro teste funcionando, podemos continuar adicionando novos casos de teste, por exemplo, para exercitar algumas das regras de validação que tenha a transação, podemos ver por exemplo, que na transação Attraction ocorre um erro quando o nome da atração é vazio, portanto, poderíamos criar um caso de teste que exercite esta regra. Aqui já adicionamos um novo grupo em nosso DataProvider dos casos de teste para testar este novo caso.

Deixamos então o nome da atração vazio, no código de resultado indicamos que esperamos que ele falhe, na mensagem colocamos a mensagem da regra de erro que está retornando a transação – neste caso o BC – e na mensagem de erro indicamos que esperamos que falhe quando o nome da atração estiver em branco

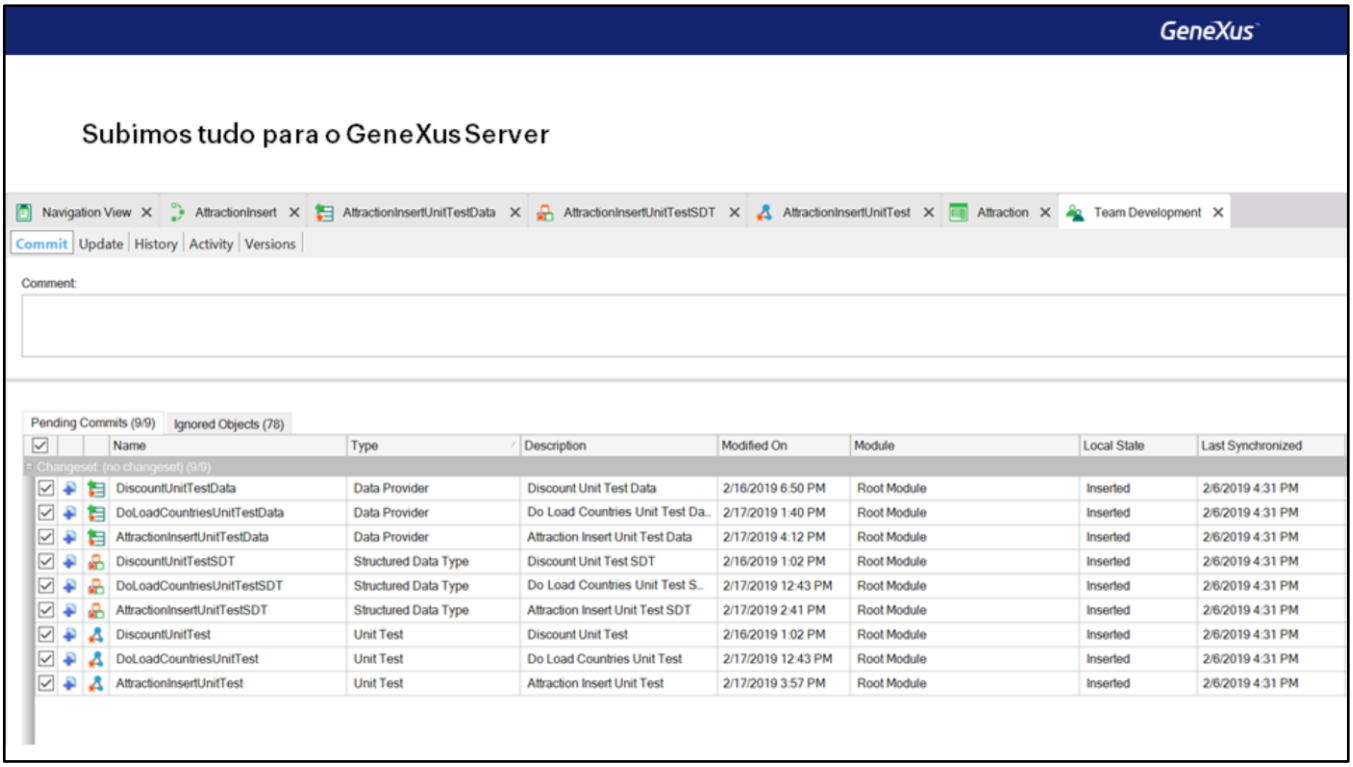
Vamos executar então nosso teste

Algo que já podemos ver enquanto o nosso teste é executado, é que entre nossos Assert temos um For Each que sempre espera que uma atração seja registrada e isto nem sempre será o caso. Em síntese, neste segundo caso de teste, na realidade esperamos que o BC gere um erro e não faça uma inserção, portanto, já podemos saber que, como o nosso teste não está correto, gerará um falso positivo, ou seja, indicará um erro onde realmente não há

O que temos que fazer é consertar nosso teste unitário para que faça os Assert corretos, em suma validaremos os dados na tabela apenas para o caso em que a inserção tenha sido bem-sucedida e iremos forçar uma falha caso haja um registro na base de dados quando o código do

resultado do evento insert não tenha sido ok,
Por outro lado, vamos forçar a falha no when none somente quando o código de resultado da inserção for OK, pois nesse caso esperamos que o registro exista.

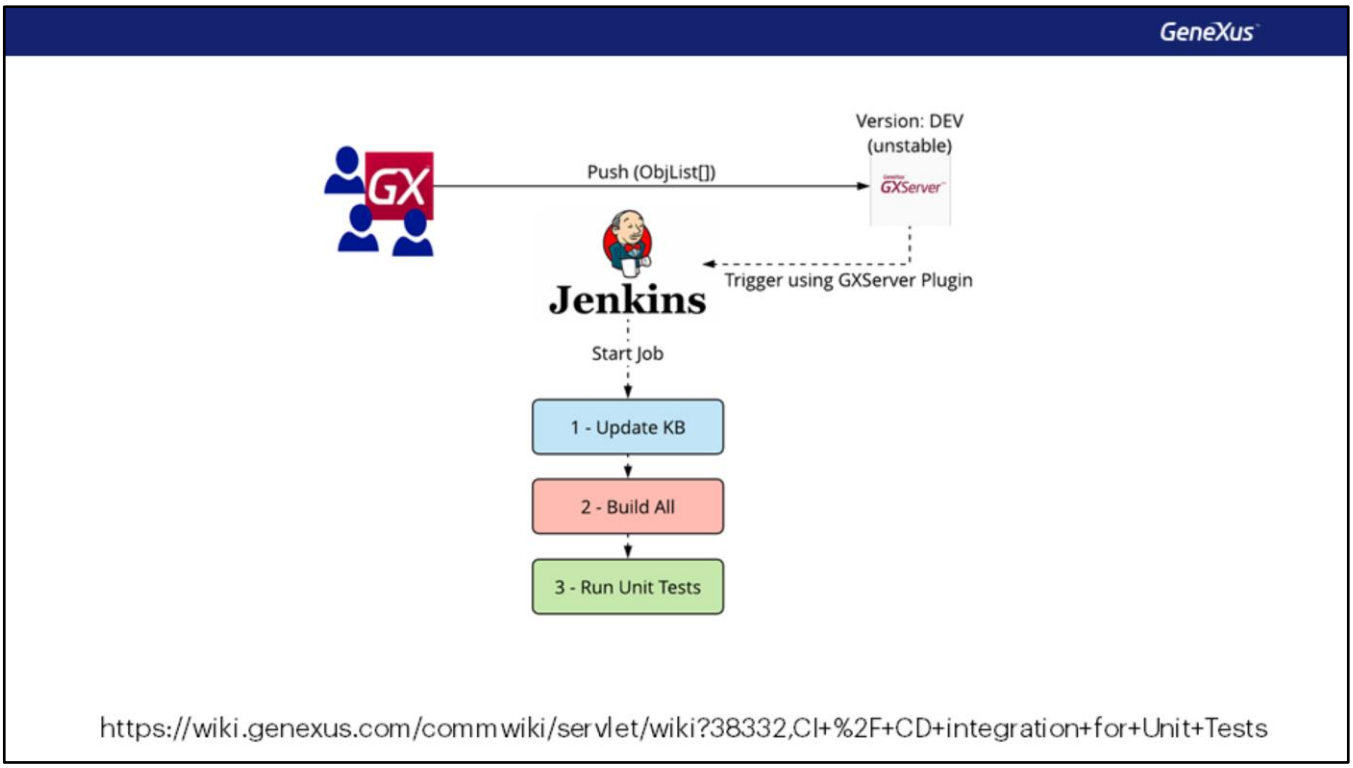
Aqui estamos fazendo o IF contra o código de resultado obtido ao invocar AttracionInsert mas teria sido igual fazê-lo com o código de resultado esperado, já que anteriormente fizemos um Assert validando que eles eram iguais e no caso de não serem, nosso teste já havia falhado por aí.



Até agora, vimos alguns exemplos de testes sobre procedimentos que interagem com a base de dados e vimos o processo pelo qual iríamos criando e revisando nossos testes.

Desta forma, nos ajuda enquanto desenvolvemos, a validar a funcionalidade que estamos implementando. Mas ao contrário de quando fazíamos alguma tela ou mensagens no console para testar, o tempo investido no desenvolvimento dos testes unitários capitaliza nossa KB já que os testes unitários poderão continuar sendo executados de forma repetitiva – constituindo parte dos testes de regressão.

Os testes podem ser enviados para o GeneXus Server e podem ser executados em diferentes ambientes – desde que a base de dados seja conhecida ou estejamos utilizando mocking –. Observe que, quando os testes são compartilhados no servidor, é necessária uma licença de GxTest. Para mais informações sobre licenciamento, consulte o Wiki.



O benefício de compartilhar os testes e tê-los no servidor, é que podemos automatizar a execução deles como parte do processo de Integração Contínua no Jenkins, a fim de evitar a publicação de uma versão se os erros forem descobertos e alertar sobre eles rapidamente para a equipe, para que o problema possa ser resolvido precocemente e, portanto, com menor custo.

Você pode ver em nosso wiki mais informações sobre Integração Contínua e como integrar os testes unitários no Jenkins.

GeneXus™

The power of doing.

Videos	training.genexus.com
Documentation	wiki.genexus.com
Certifications	training.genexus.com/certifications