**Requirement 3** - The system must be able to add new inventory items (products) with no duplicate records.

The following code creates a stored procedure to add new items. It validates that the product does already exist in the system and that the product does have a valid type.

drop procedure sp\_add\_product

go

create procedure sp\_add\_product (@pro\_name varchar(30), @pro\_instock int, @ty\_id int, @pro\_base numeric(5,2)) as

begin

--Checks for duplicate product names

if exists (select \* from t\_product where upper(replace(pro\_name, ' ', ''))=upper(replace(@pro\_name, ' ', '')))

begin

select @pro\_name, 'already exists in this system as a product.'

return

end

else

--Checks that type is valid

if not exists (select \* from t\_type where ty\_id=@ty\_id)

begin

select @pro\_name, 'was not given a valid type.'

return

end

--Inserts product data into t\_product table

begin transaction

insert into t\_product

(pro\_name, pro\_instock, ty\_id, pro\_base)

values

(@pro\_name, @pro\_instock, @ty\_id, @pro\_base)

if @@error<>0

begin

rollback transaction

select 'Product ', @pro\_name, ' not added'

return

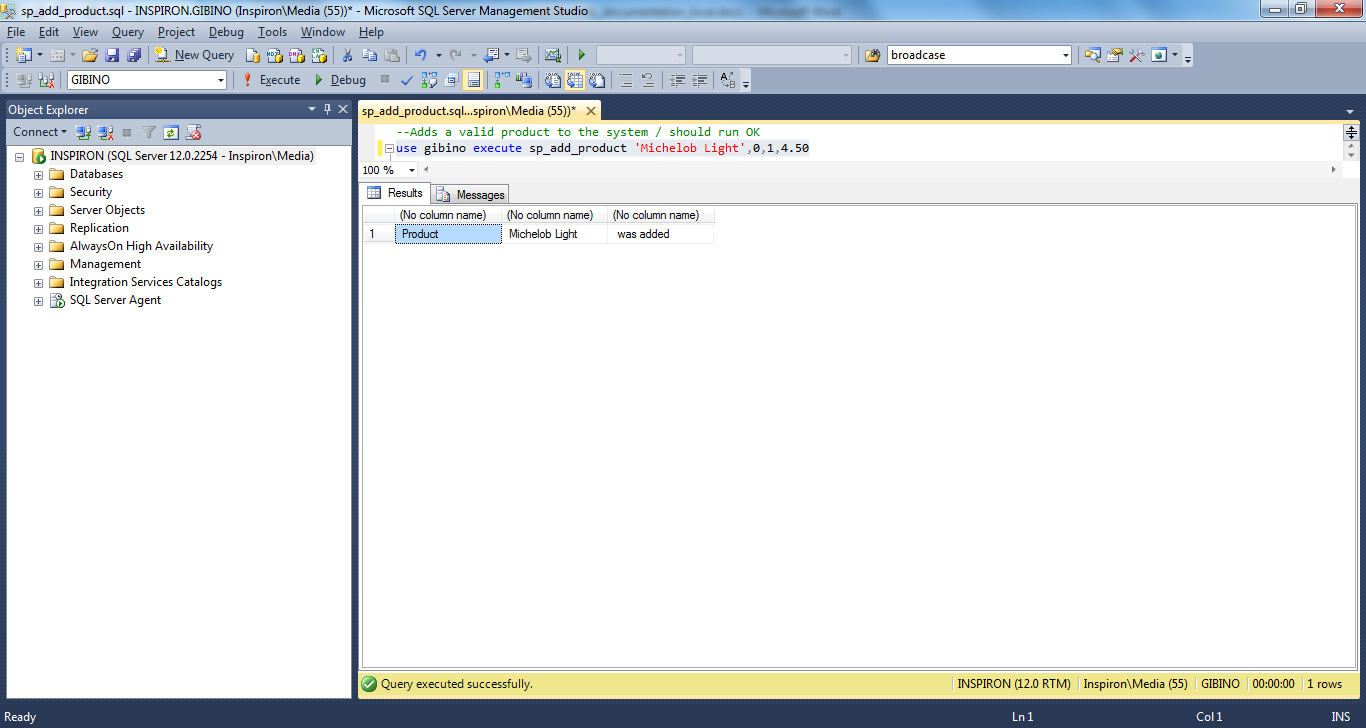
end

commit transaction

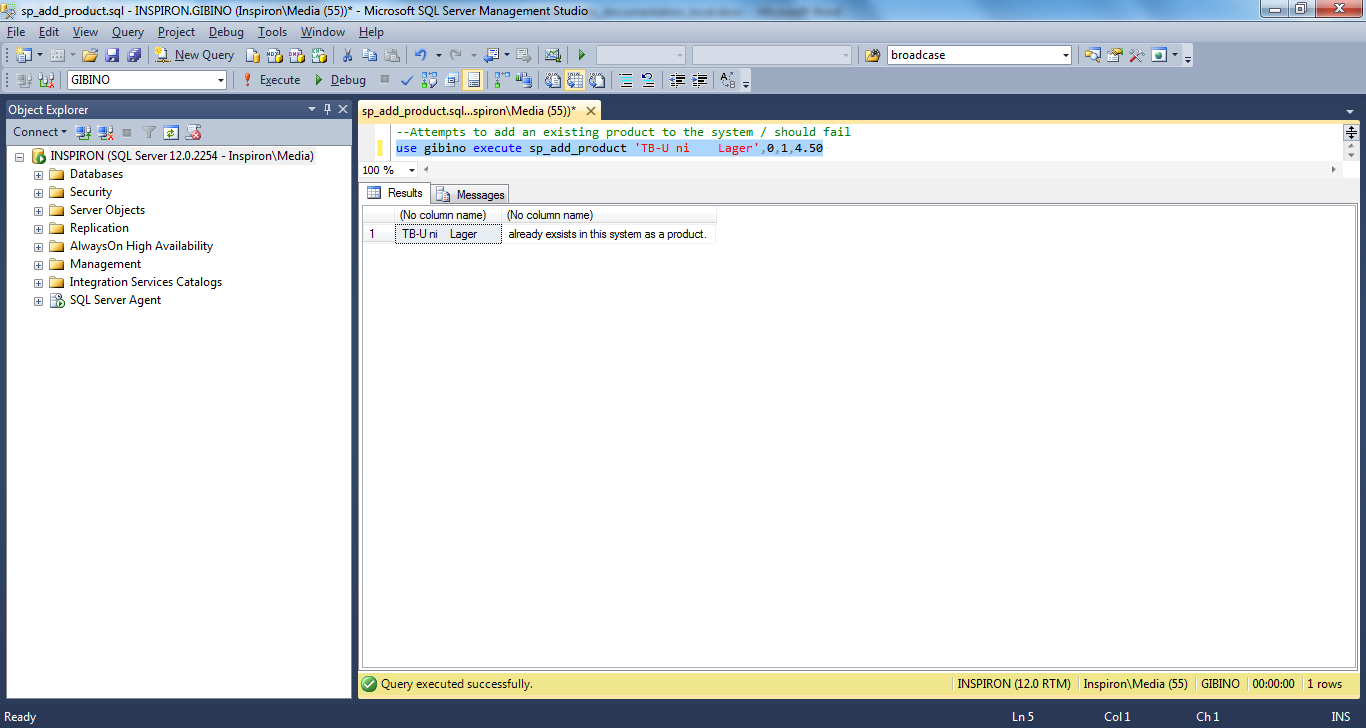
select 'Product ', @pro\_name, ' was added'

end

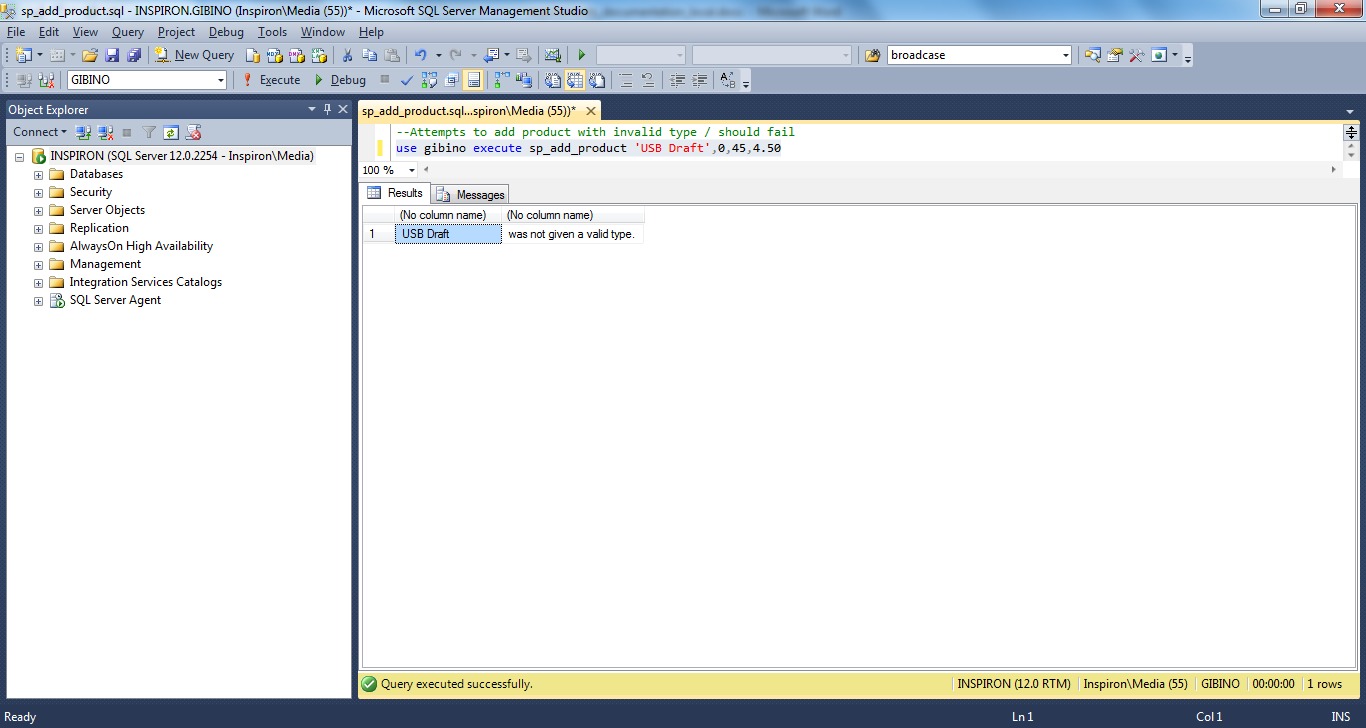
This screen shot demonstrates the successful insertion of a product.



This screen shot demonstrates a failed attempt to insert an existing product.



This screen shot demonstrates a failed attempt to insert a product with an invalid type.



**Requirement 4** - The system must be able to add new vendors.

The following code creates a stored procedure to add new vendors. It validates that the vendor’s name does not already exist in the system.

drop procedure sp\_add\_vendor

go

create procedure sp\_add\_vendor

(@ven\_name varchar(30),@ven\_street1 varchar(30), @ven\_street2 varchar(30), @ven\_city varchar(30) , @ven\_state char(2),

@ven\_zip numeric(5,0), @ven\_phone varchar(12), @ven\_email varchar(50),@ven\_contact varchar(50))

as

begin

--Checks for duplicate vendor names

if exists (select \* from t\_vendor where upper(replace(ven\_name, ' ' , '')) = upper(replace(@ven\_name, ' ','')))

begin

select @ven\_name, 'already exists in this system as a vendor'

return

end

--Inserts vendor information into t\_vendor table

begin transaction

insert into t\_vendor

(ven\_name,ven\_street1, ven\_street2, ven\_city, ven\_state,

ven\_zip, ven\_phone, ven\_email, ven\_contact)

values

(@ven\_name,@ven\_street1, @ven\_street2, @ven\_city, @ven\_state,

@ven\_zip, @ven\_phone, @ven\_email, @ven\_contact)

if @@error<>0

begin

rollback transaction

select 'Vendor ', @ven\_name, ' not added'

return

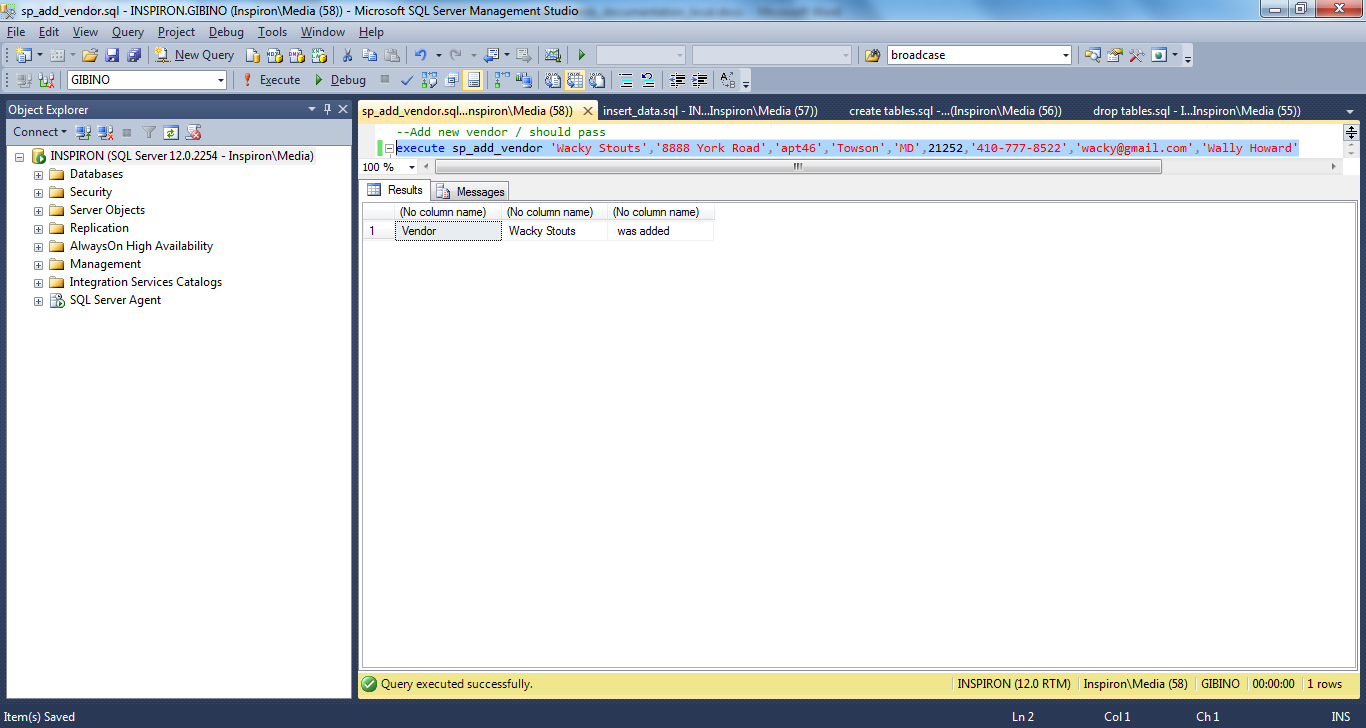
end

commit transaction

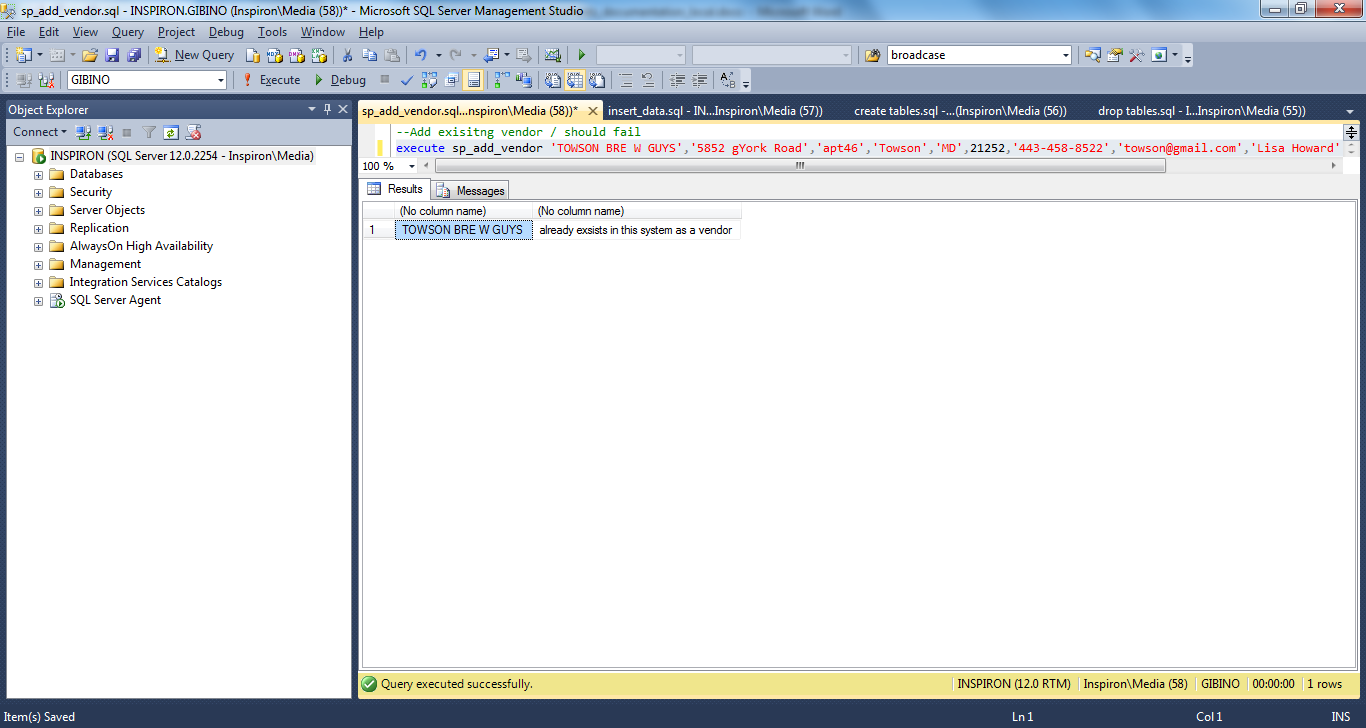
select 'Vendor ', @ven\_name, ' was added'

end

This screen shot demonstrates the successful insertion of a new vendor.



This screen shot demonstrates a failed attempt to insert a new vendor with an existing name.



**Requirement 5** - The system must be able to display the beverage inventory belonging to a given vendor.

The following code creates a stored procedure which can be used to query the inventory for a given vendor. It validates that the vendor exists in the system.

drop procedure sp\_inventory\_by\_vendor

go

create procedure sp\_inventory\_by\_vendor

(@ven\_id int)

as begin

--Checks that vendor exists

if exists (select \* from t\_vendor where ven\_id=@ven\_id)

begin

select v.ven\_name as Vendor, p.pro\_name as Product, p.pro\_instock as Inventory

from t\_vendor v left join t\_product p on v.ven\_id = p.ven\_id

where v.ven\_id=@ven\_id

order by Vendor, Product

return

end

else

begin

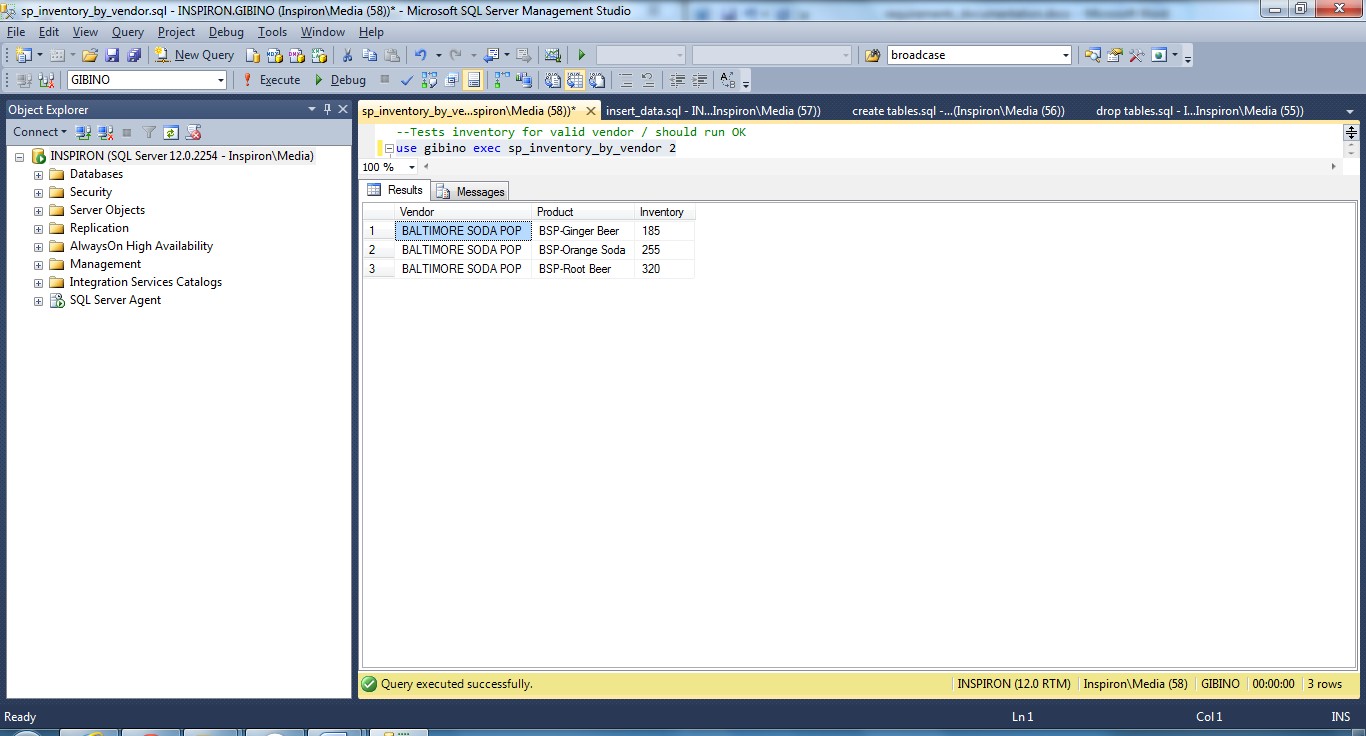
select @ven\_id, 'is not a valid vendor.'

return

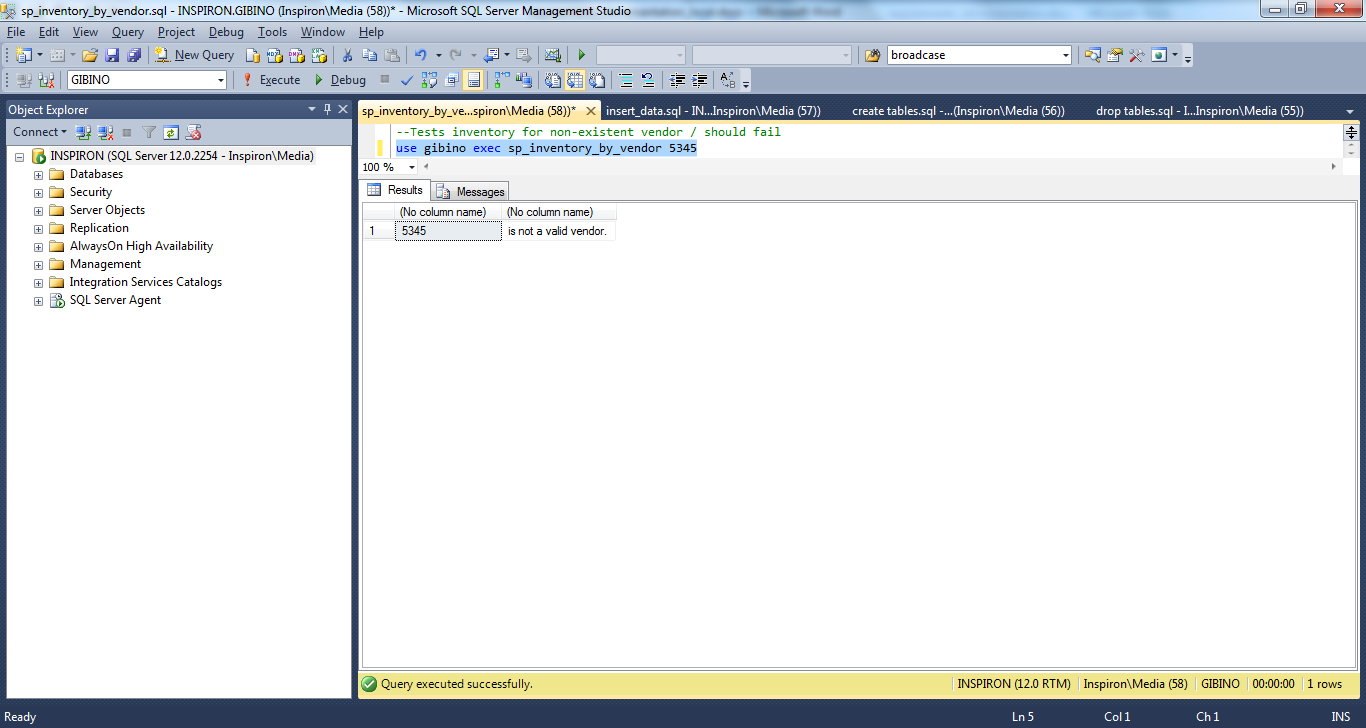
end

end

This screen shot demonstrates the stored procedure pulling the inventory for a given vendor.



This screenshot demonstrates the error that occurs when the stored procedure is run for a non-existent vendor.



**Requirement 6** - The system must produce a formatted report of all products and available quantities for every vendor.

This stored procedure uses nested cursors to produce a formatted report of products and inventories, by vendor.

drop proc sp\_inventory\_cursoring

go

--This stored procedure uses two nested cursors to report on each vendor and the available products and inventory

--The first cursor (vendor\_cursor) selects the vendors, one-by-one

--The second cursors (product\_cursors) selects the products (beverages) and available inventory for each product

--for the current vendor in the vendor\_cursor

create proc sp\_inventory\_cursoring

as begin

set nocount on

--The various variables are declared. The length of the broadcase variable is set to 140, in case this cursor

--will be used for a twitter feed.

declare

@ven\_id int, @ven\_name varchar(30), @broadcast varchar(140), @pro\_name varchar(30), @pro\_instock int

--This is the title that will be presented at the top of the report

print 'INVENTORY REPORT, LISTS ALL VENDORS AND AVAILABLE QUANTITIES OF EACH BEVERAGE'

declare vendor\_cursor cursor for

select ven\_id, ven\_name

from t\_vendor

order by ven\_name

open vendor\_cursor

fetch next from vendor\_cursor

into @ven\_id, @ven\_name

while @@fetch\_status=0

begin

print ' '

select @broadcast = '---Beverages from Vendor ' + @ven\_name

print @broadcast

-- The product\_cursor, nested inside of the vendor\_cursor begins here

declare product\_cursor cursor for

select p.pro\_name, p.pro\_instock

from t\_product p

where p.ven\_id=@ven\_id

order by p.pro\_name

open product\_cursor

fetch next from product\_cursor into @pro\_name, @pro\_instock

if @@fetch\_status <> 0

print ' This vendor does not have any products in the current inventory.'

while @@fetch\_status=0

begin

--Had to cast the integer variable, pro\_instock, into a varchar to concatenate it.

select @broadcast = ' ' + @pro\_name + ', ' + cast(@pro\_instock as varchar) + ' units in stock.'

print @broadcast

fetch next from product\_cursor into @pro\_name, @pro\_instock

--Closes the product\_cursor for the current vendor

end

close product\_cursor

deallocate product\_cursor

--Vendor\_cursor moves to the next vendor the next vendor

fetch next from vendor\_cursor

into @ven\_id, @ven\_name

end

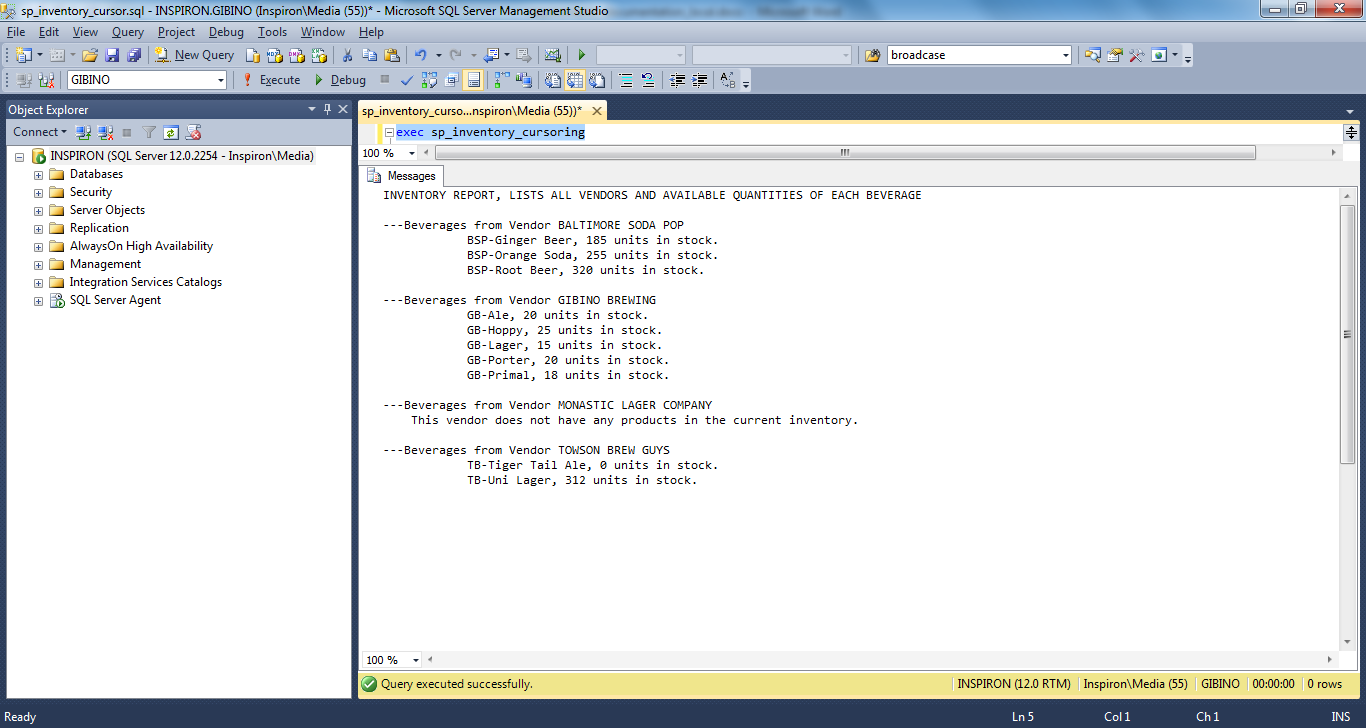
--Vendor cursor closes

close vendor\_cursor

deallocate vendor\_cursor

end

This screenshot shows the successful execution of the report.



**Requirement 7** – The system must display original prices and current prices

**Requirement 8** – The system must display % difference in price from current to original.

**Requirement 9** – The system must display daily high price for beer.

**Requirement 10** – The system must display daily low price for beer.

The following code creates a stored procedure view the Price Report data with a date parameter.

/\* CHECK IF PROCEDURE EXISTS, DROP AND RECREATE IT \*/

IF OBJECTPROPERTY(object\_id('dbo.sp\_price\_report'), N'IsProcedure') = 1

DROP PROCEDURE [dbo].[sp\_price\_report]

GO

CREATE PROCEDURE dbo.sp\_price\_report

@date DATETIME

AS

BEGIN

BEGIN TRANSACTION

/\* HELPER TABLE \*/

IF OBJECT\_ID('dbo.t\_price\_diff', 'U') IS NOT NULL

DROP TABLE dbo.t\_price\_diff

CREATE TABLE dbo.t\_price\_diff

(

diff\_id INT IDENTITY(1,1) PRIMARY KEY,

pro\_id INT FOREIGN KEY REFERENCES dbo.t\_product(pro\_id),

diff\_perc DECIMAL(5,1) NOT null

);

/\* CALCULATE THE PRICE DIFFERENCES \*/

INSERT INTO T\_PRICE\_DIFF (PRO\_ID, DIFF\_PERC)

SELECT pri.pro\_id, ((pri.pro\_price - pro.pro\_base)/pro.pro\_base) \* 100

FROM t\_price pri INNER JOIN t\_product pro

ON pri.pro\_id = pro.pro\_id;

/\* ROLLBACK ON ERROR \*/

IF @@error <> 0

BEGIN

ROLLBACK TRANSACTION

SELECT ' There was a problem creating the price report'

RETURN

END

COMMIT TRANSACTION;

/\* QUERY FOR THE REPORT \*/

SELECT pro.pro\_name AS Product, pro.pro\_base AS OriginalPrice, pri.pro\_price AS CurrentPrice, d.diff\_perc AS PercentageDifference

, (CASE WHEN max(s.pro\_price) < pri.pro\_price THEN pri.pro\_price ELSE max(s.pro\_price) END) AS DailyHigh

, min(s.pro\_price) AS DailyLow

FROM t\_price pri

INNER JOIN t\_product pro ON pri.pro\_id = pro.pro\_id

INNER JOIN t\_price\_diff d ON d.pro\_id = pri.pro\_id

INNER JOIN t\_pos\_sales s ON s.pro\_id = d.pro\_id

WHERE DAY(s.pos\_datetime) = DAY(@date)

GROUP BY pri.pro\_id, pro.pro\_name,pro.pro\_base, pri.pro\_price, d.diff\_perc

UNION

/\* ADD PRODUCTS WITH NO SALES \*/

SELECT pro.pro\_name AS Product, pro.pro\_base AS OriginalPrice, pri.pro\_price AS CurrentPrice, 0 AS PercentageDifference, pro.pro\_base AS DailyHigh, pro.pro\_base AS DailyLow

FROM t\_price pri

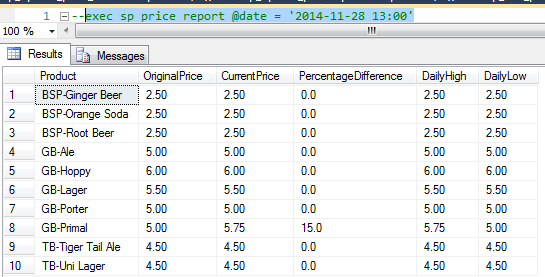
INNER JOIN t\_product pro ON pri.pro\_id = pro.pro\_id

INNER JOIN t\_price\_diff d ON d.pro\_id = pri.pro\_id

WHERE pri.pro\_id NOT IN (SELECT pro\_id FROM t\_pos\_sales WHERE DAY(pos\_datetime) = DAY(@date));

END

This screen capture demonstrates successful generation of a Price Report for the day input.



**Requirement 11** – The system must display prices for most recently sold beers.

The following code will create a stored procedure to return the products sold within the last hour, the price they sold for and the time of sale.

/\* CHECK IF PROCEDURE EXISTS, DROP AND RECREATE IT \*/

IF OBJECTPROPERTY(object\_id('dbo.sp\_recent\_sales'), N'IsProcedure') = 1

DROP PROCEDURE [dbo].[sp\_recent\_sales]

GO

CREATE PROCEDURE dbo.sp\_recent\_sales

AS

BEGIN

SELECT pro\_name AS Product, pro\_price AS PriceSold, pos\_datetime AS TimeOfSale

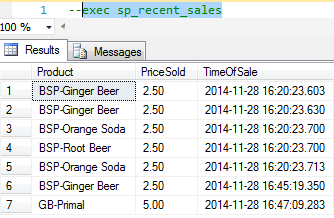
FROM t\_pos\_sales s

INNER JOIN t\_product p ON p.pro\_id = s.pro\_id

WHERE pos\_datetime > DateAdd(Hour, -1, GETDATE()) and pos\_datetime < GETDATE();

END

This screen capture demonstrates the successful generation of a recent sales report.



**Requirement 12** – The system must display top selling products of the day.

The following code will create a stored procedure to generate a report of the top 3 products for the day.

/\* CHECK IF PROCEDURE EXISTS, DROP AND RECREATE IT \*/

IF OBJECTPROPERTY(object\_id('dbo.sp\_top\_sellers'), N'IsProcedure') = 1

DROP PROCEDURE [dbo].[sp\_top\_sellers]

GO

CREATE PROCEDURE dbo.sp\_top\_sellers

AS

BEGIN

/\* SELECT TOP 3 SELLING PRODUCTS FOR THE CURRENT DAY \*/

SELECT top 3 pro\_name, max(pos\_qty) AS TopSeller

FROM t\_pos\_sales s

INNER JOIN t\_product p ON p.pro\_id = s.pro\_id

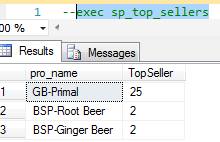
WHERE Day(pos\_datetime) = Day(GetDate())

GROUP BY p.pro\_id, pro\_name

ORDER BY TopSeller DESC;

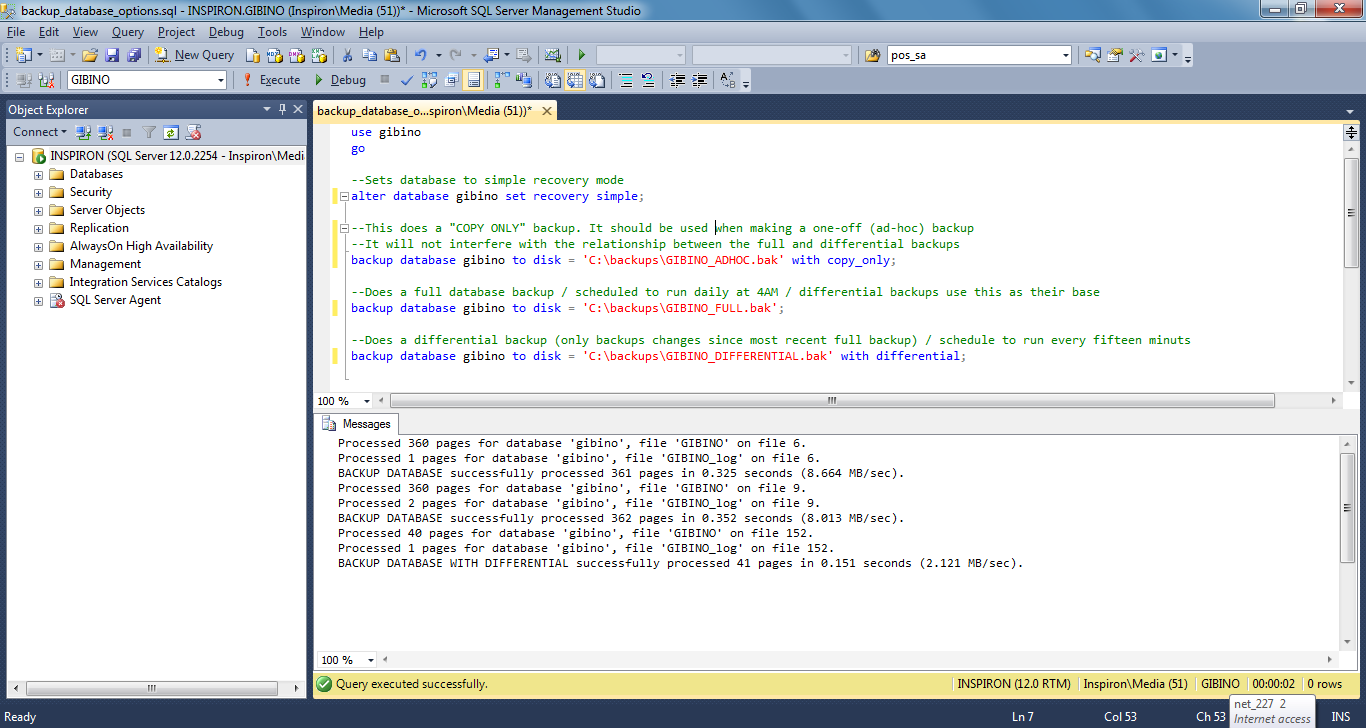
END

This screen capture demonstrates successful generation of a top sellers report.

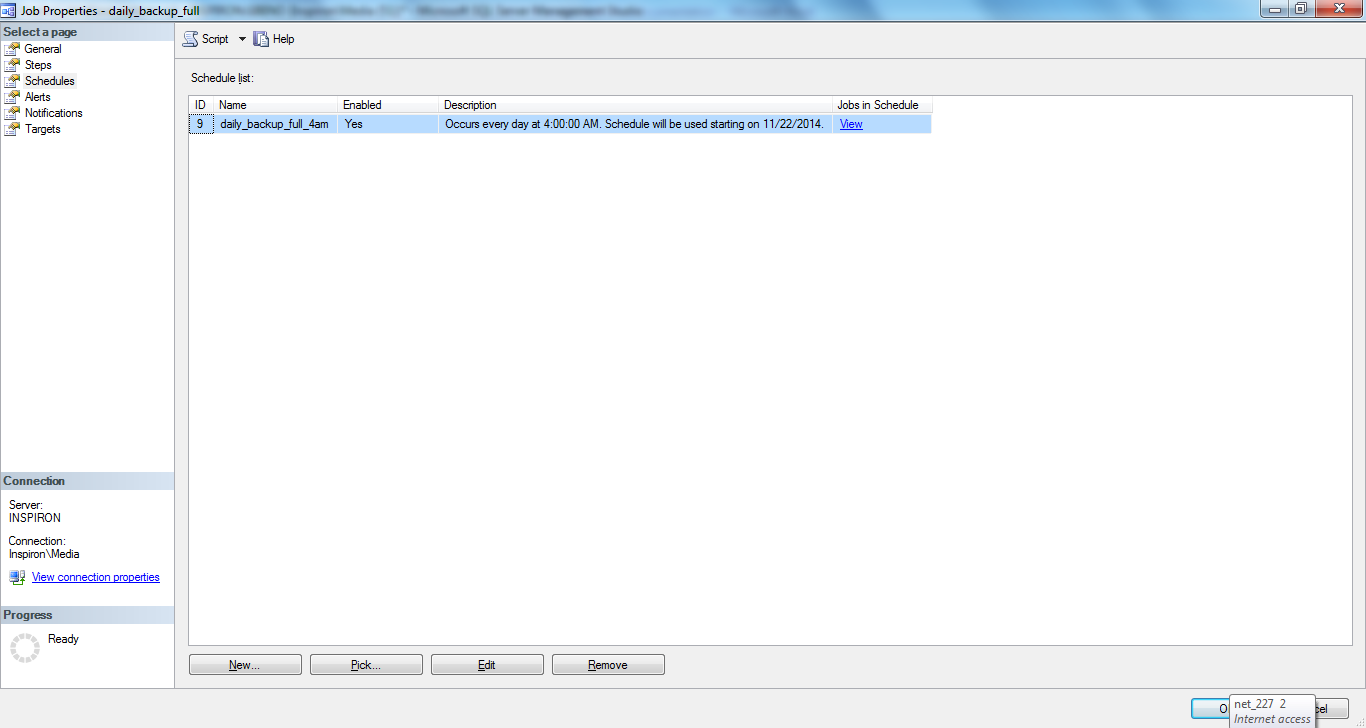


**Requirement 14** -The system must create a nightly backup, after normal business hours.

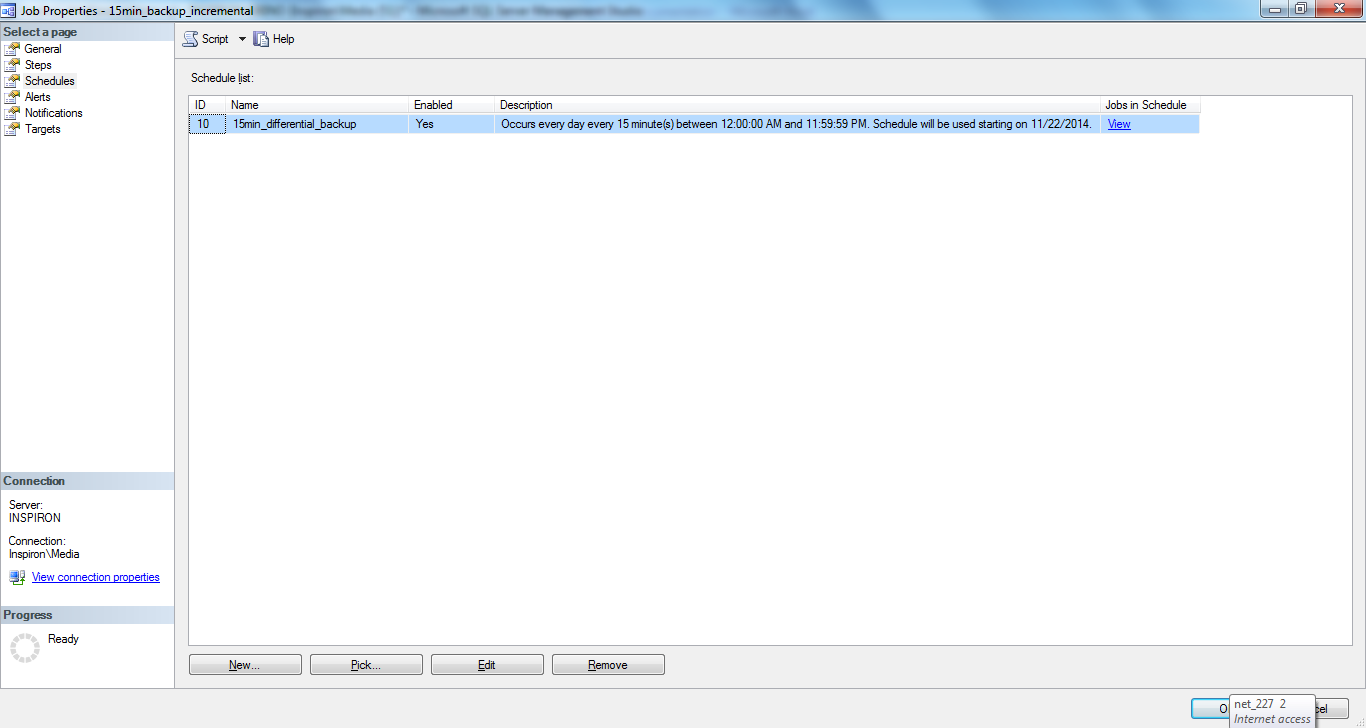
The following code demonstrates how to setup the database for ad hoc, full, and differential backups. It also shows how to put the database into simple recovery mode.



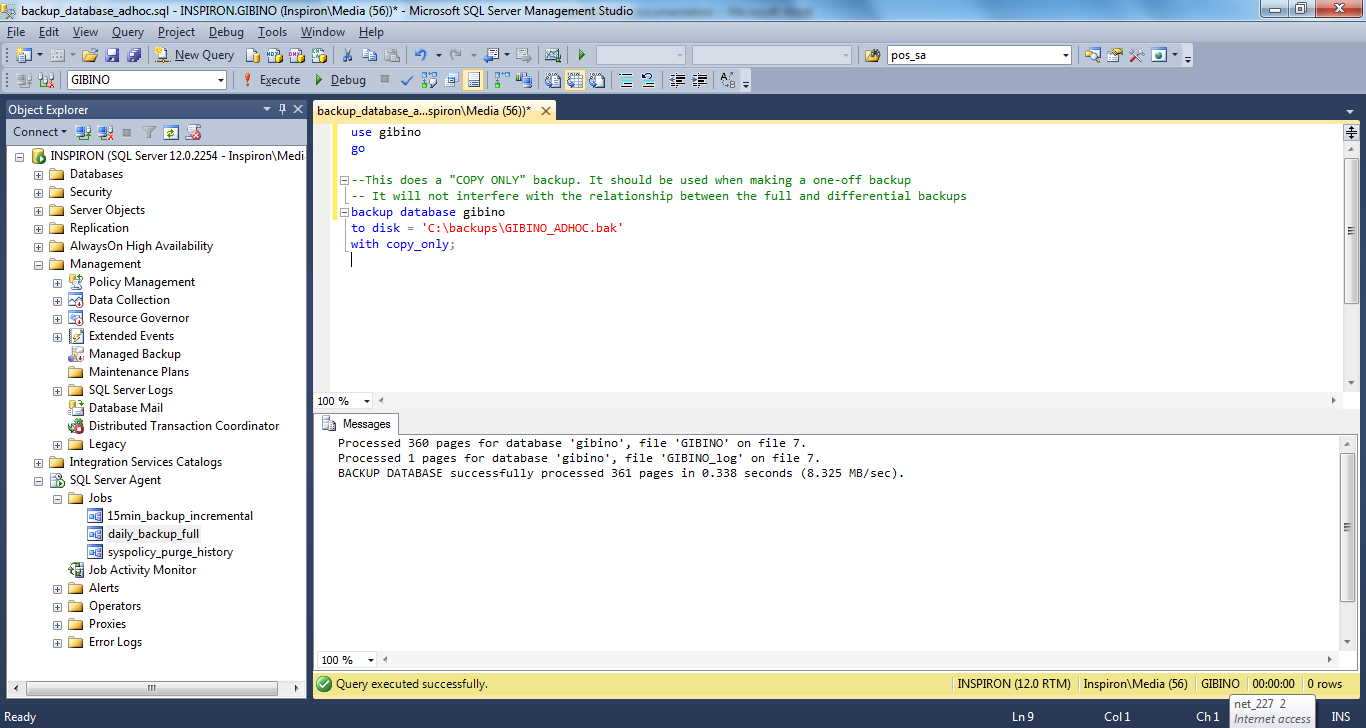
This screen capture shows that the full backup is a scheduled job, which runs at 4:00 AM every day.



This screen capture shows that the differential backup is a scheduled job, which runs every 15 minutes.



This screen capture demonstrates the code and output for a successful ad hoc backup.



**Requirement 15** – The system must be able to update product prices of based on sales.

The following code will create a stored procedure to update product prices based on sales.

/\* CHECK IF PROCEDURE EXISTS, DROP AND RECREATE IT \*/

IF OBJECTPROPERTY(object\_id('dbo.sp\_update\_price\_from\_sales'), N'IsProcedure') = 1

DROP PROCEDURE [dbo].[sp\_update\_price\_from\_sales]

GO

CREATE PROCEDURE dbo.sp\_update\_price\_from\_sales

/\* DATE PARAMETERS \*/

@start\_time DATETIME,

@end\_time DATETIME

AS

BEGIN

BEGIN TRANSACTION;

/\* HELPER TABLE \*/

IF OBJECT\_ID('dbo.t\_sales\_info', 'U') IS NOT NULL

DROP TABLE dbo.t\_sales\_info

CREATE TABLE dbo.t\_sales\_info

(

sal\_id INT IDENTITY(1,1) PRIMARY KEY,

pro\_id INT FOREIGN KEY REFERENCES dbo.t\_product(pro\_id),

qty\_sold INT NOT null

);

/\* GRAB ALL THE SALES FOR THE TIME PERIOD \*/

INSERT INTO T\_SALES\_INFO (PRO\_ID, QTY\_SOLD)

SELECT b.pro\_id,sum(b.qty\_sold)

FROM v\_beer\_sales b

WHERE time\_of\_sale between @start\_time AND @end\_time

GROUP BY b.pro\_id

/\* HELPER TABLE \*/

IF OBJECT\_ID('dbo.t\_sales\_perc', 'U') IS NOT NULL

DROP TABLE dbo.t\_sales\_perc

CREATE TABLE dbo.t\_sales\_perc

(

sal\_id INT IDENTITY(1,1) PRIMARY KEY,

pro\_id INT FOREIGN KEY REFERENCES dbo.t\_product(pro\_id),

pct\_of\_sales INT NOT null

);

/\* PERCENTAGE OF SALES MATH \*/

INSERT INTO T\_SALES\_PERC (PRO\_ID, PCT\_OF\_SALES)

SELECT s.pro\_id, ROUND(s.qty\_sold \* 100.0/(SELECT sum(s.qty\_sold) FROM t\_sales\_info s),2)

FROM t\_sales\_info s

/\* HELPER TABLE \*/

IF OBJECT\_ID('dbo.t\_price\_adjust', 'U') IS NOT NULL

DROP TABLE dbo.t\_price\_adjust

CREATE TABLE dbo.t\_price\_adjust

(

pa\_id INT IDENTITY(1,1) PRIMARY KEY,

pro\_id INT FOREIGN KEY REFERENCES dbo.t\_product(pro\_id),

price\_adjust DECIMAL(5,2) NOT null

);

/\* PRICE ADJUSTMENT CALCULATIONS \*/

INSERT INTO T\_PRICE\_ADJUST(PRO\_ID, PRICE\_ADJUST)

SELECT p.pro\_id, CASE

WHEN pct\_of\_sales = 0 THEN 0

WHEN pct\_of\_sales > 0 AND pct\_of\_sales < 20 THEN 0

WHEN pct\_of\_sales >= 20 AND pct\_of\_sales < 40 THEN .25

WHEN pct\_of\_sales >= 40 AND pct\_of\_sales < 60 THEN .5

WHEN pct\_of\_sales >= 60 AND pct\_of\_sales < 80 THEN .75

WHEN pct\_of\_sales >= 80 AND pct\_of\_sales <= 100 THEN 1

END

FROM t\_sales\_perc p;

/\* UPDATE T\_PRICE TABLE \*/

UPDATE p

SET p.pro\_price = p.pro\_price + pa.price\_adjust

FROM T\_PRICE p INNER JOIN T\_PRICE\_ADJUST pa

ON p.pro\_id = pa.pro\_id;

/\* ROLLBACK IF THERE WAS AN ERROR \*/

IF @@error <> 0

BEGIN

ROLLBACK TRANSACTION

SELECT ' Price update was unsuccessful'

RETURN

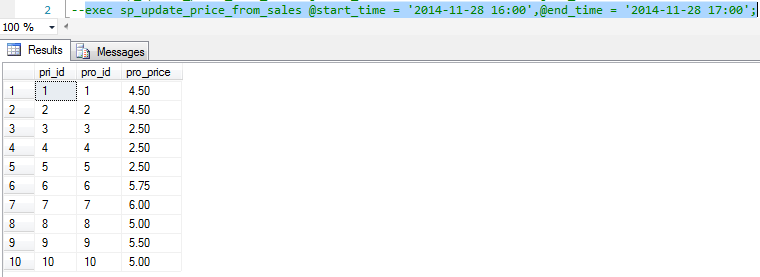
END

COMMIT TRANSACTION;

SELECT \* FROM T\_PRICE;

END

This screen capture demonstrates successful update of product prices.



**Requirement 16** – The system must be update real-time inventory levels according to sales

**Requirement 19** – The system must not be able to sell a quantity which exceeds the inventory available.

**Requirement 20** – The system must not be able to sell alcoholic products to a customer below the age of 21

The following code will create a stored procedure to create point of sale transactions. Once a sale is made and entry to the table is created and inventory levels are updated. There is validation for alcoholic products.

/\* CHECK IF PROCEDURE EXISTS, DROP AND RECREATE IT \*/

IF OBJECTPROPERTY(object\_id('dbo.sp\_add\_pos\_sale'), N'IsProcedure') = 1

DROP PROCEDURE [dbo].[sp\_add\_pos\_sale]

GO

CREATE PROCEDURE dbo.sp\_add\_pos\_sale

@pro\_id INT,

@pur\_qty INT,

@cus\_id INT,

@pos\_paid INT

AS

BEGIN

/\* VALIDATE PRODUCT \*/

IF NOT EXISTS(SELECT \* FROM t\_product WHERE pro\_id = @pro\_id)

BEGIN

SELECT 'This product does NOT exist!'

RETURN

END

/\* VALIDATE CUSTOMER \*/

IF NOT EXISTS(SELECT \* FROM t\_customer WHERE cus\_id = @cus\_id)

BEGIN

SELECT 'This customer does NOT exist!'

RETURN

END

/\* VALIDATE CUSTOMER \*/

IF @pur\_qty <= 0

BEGIN

SELECT 'This is NOT a valid purchase amount!'

RETURN

END

/\* VALIDATE SUFFICIENT INVENTORY \*/

IF NOT EXISTS(SELECT \* FROM dbo.t\_product WHERE pro\_id=@pro\_id AND pro\_instock>=@pur\_qty)

BEGIN

SELECT 'We do NOT have enough of this product IN stock. The amount IN stock is', (SELECT pro\_instock FROM t\_product WHERE pro\_id=@pro\_id)

RETURN

END

/\* VALIDATE AGE FOR RESTRICTED ITEM \*/

IF EXISTS(SELECT \* FROM t\_product WHERE pro\_id=@pro\_id AND ty\_id IN (SELECT ty\_id FROM t\_type WHERE ty\_restricted=1))

BEGIN

/\* CHECK FOR CUSTOMER AGE, CREATE A HELPER TABLE TO DETERMINE AGE \*/

BEGIN TRANSACTION

IF OBJECT\_ID('dbo.t\_cus\_age', 'U') IS NOT NULL

DROP TABLE dbo.t\_cus\_age

CREATE TABLE dbo.t\_cus\_age

(

cage\_id INT IDENTITY(1,1) PRIMARY KEY,

cus\_id INT FOREIGN KEY REFERENCES dbo.t\_product(pro\_id),

age INT NOT null

);

INSERT INTO T\_CUS\_AGE(CUS\_ID, AGE)

SELECT cus\_id, DATEDIFF(hour, cus\_dob, GETDATE())/8766 AS AgeInYears FROM t\_customer;

/\* CUSTOMER IS NOT OF AGE \*/

IF NOT EXISTS(SELECT \* FROM t\_cus\_age a

INNER JOIN t\_customer c ON a.cus\_id = c.cus\_id

WHERE age < 21 AND a.cus\_id = @cus\_id)

BEGIN

ROLLBACK TRANSACTION

SELECT 'Alcoholic beverages can only be purchased BY customers who are 21 years of age or older'

RETURN

END

COMMIT TRANSACTION

END

BEGIN TRANSACTION

/\* CREATE A SALE \*/

insert into dbo.t\_pos\_sales(pos\_qty, cus\_id, pro\_id, pos\_paid, pro\_price)

values(@pur\_qty, @cus\_id, @pro\_id, @pos\_paid, (SELECT pro\_price FROM dbo.t\_price WHERE pro\_id = @pro\_id));

update t\_product

set pro\_instock = (pro\_instock - @pur\_qty)

WHERE pro\_id = @pro\_id;

/\* ROLLBACK IF AN ERROR HAS OCCURRED \*/

IF @@error <> 0

BEGIN

ROLLBACK TRANSACTION

SELECT ' Sale was NOT completed'

RETURN

END

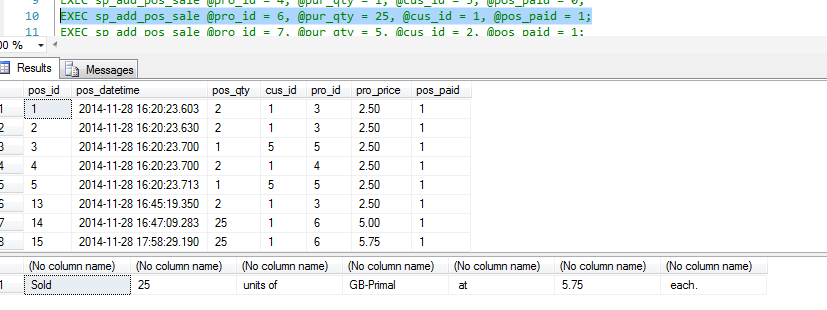
COMMIT TRANSACTION;

SELECT \* FROM t\_pos\_sales;

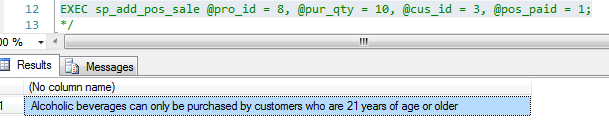
SELECT 'Sold ', @pur\_qty, 'units of ', (SELECT pro\_name FROM t\_product WHERE pro\_id = @pro\_id), ' at ', (SELECT pro\_price FROM t\_price WHERE pro\_id = @pro\_id), ' each.'

END

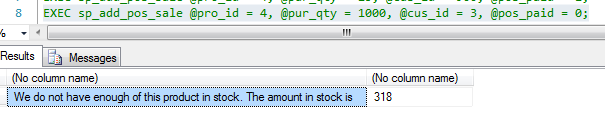
This screen capture demonstrates a valid purchase.



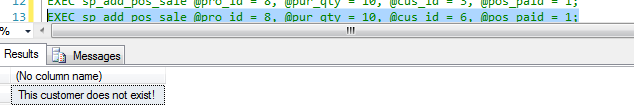
This screen capture demonstrates a customer under 21 cannot purchase an alcoholic beverage.



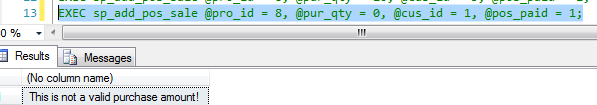
This screen capture demonstrates validation for available inventory.



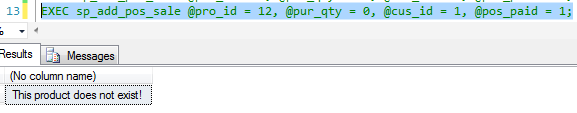
This screen capture demonstrates validation for an existing customer.



This screen capture demonstrates validation for a valid purchase amount (greater than 0)



This screen capture demonstrates validation for a valid product.



**Requirement 18** - The system must be able to update the vendor information.

The following code demonstrates the creation of a stored procedure, which allows the user to update the information for a given vendor. It validates that the vendor name is not updated to the name of an existing vendor.

drop procedure sp\_update\_vendor

go

create procedure sp\_update\_vendor

(@ven\_id int, @ven\_name varchar(30),@ven\_street1 varchar(30), @ven\_street2 varchar(30), @ven\_city varchar(30) , @ven\_state char(2),

@ven\_zip numeric(5,0), @ven\_phone varchar(12), @ven\_email varchar(50),@ven\_contact varchar(50))

as

begin

--Checks for that another vendor, with the same name, does not exist

if exists (select \* from t\_vendor where upper(replace(ven\_name, ' ', '' ))=upper(replace(@ven\_name, ' ', '' )) and ven\_id <> @ven\_id)

begin

select @ven\_name, 'already exists in this system as a vendor. Please check that you have the correct vendor information.'

return

end

else

--Updates vendor information in t\_vendor table

begin transaction

update t\_vendor set

ven\_name=@ven\_name, ven\_street1=@ven\_street1, ven\_street2=@ven\_street2, ven\_city=@ven\_city, ven\_state=@ven\_state, ven\_zip=@ven\_zip, ven\_phone=@ven\_phone, ven\_email=@ven\_email, ven\_contact=@ven\_contact

where ven\_id=@ven\_id

if @@error<>0

begin

rollback transaction

select 'Vendor ', @ven\_name, ' was not updated'

return

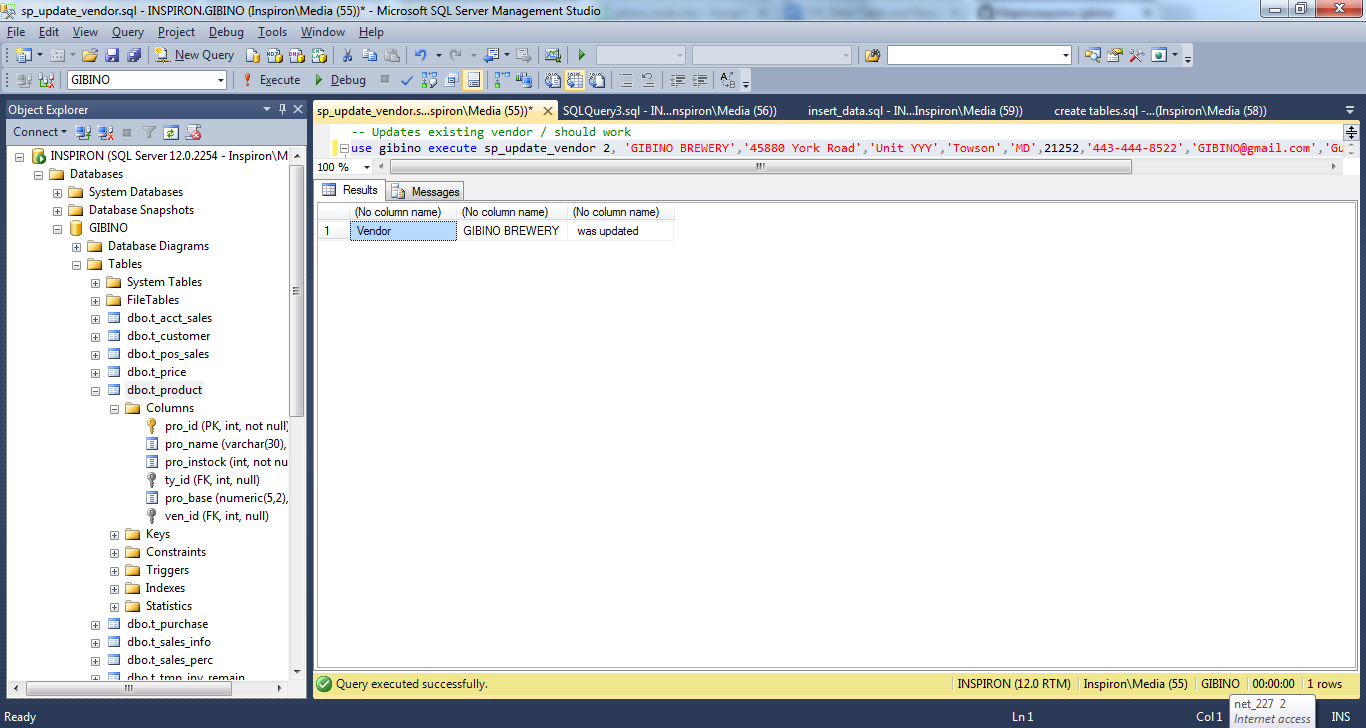
end

commit transaction

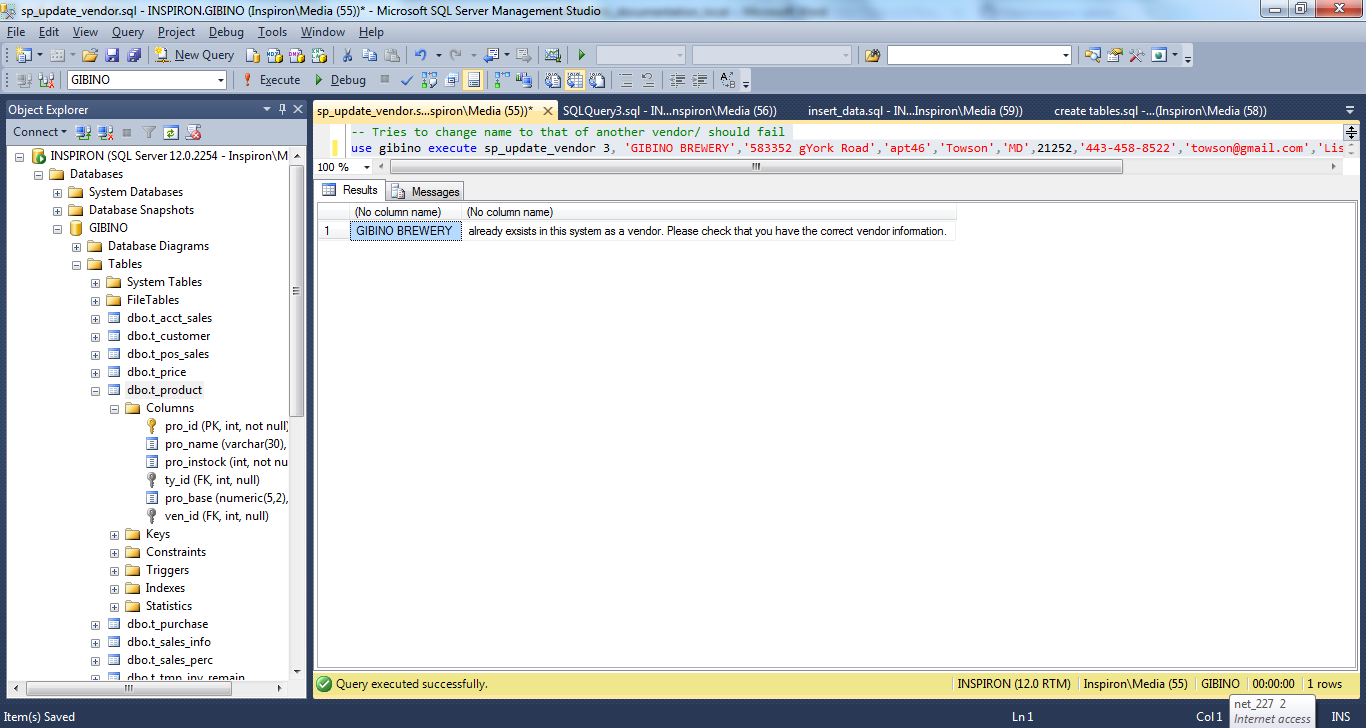
select 'Vendor ', @ven\_name, ' was updated'

end

This shows the successful name change for a vendor.



This shows the resulting error that occurs when the user attempts to update the vendor’s name to the name of another vendor.



**Requirement 21** – The system must be able to display the unpaid items for a customer.

The following code creates a stored procedure to query the system for unpaid items. It validates that the product does already exist in the system and the product does have a valid type.

/\* CHECK IF PROCEDURE EXISTS, DROP AND RECREATE IT \*/

IF OBJECTPROPERTY(object\_id('sp\_customer\_tab'), N'IsProcedure') = 1

DROP PROCEDURE [dbo].[sp\_customer\_tab]

GO

CREATE PROCEDURE dbo.sp\_customer\_tab

@cus\_id INT

AS

BEGIN

/\* ROLLBACK IF AN ERROR HAS OCCURRED \*/

IF NOT EXISTS (SELECT \* FROM T\_CUSTOMER WHERE cus\_id = @cus\_id)

BEGIN

SELECT 'Invalid Customer!'

RETURN

END

/\* ITEMIZED LIST OF UNPAID ITEMS \*/

SELECT pro.pro\_name, s.pro\_price, s.pos\_qty

FROM t\_customer c

INNER JOIN t\_pos\_sales s ON c.cus\_id=s.cus\_id

INNER JOIN t\_product pro ON pro.pro\_id=s.pro\_id

WHERE s.pos\_paid=0

AND s.cus\_id = @cus\_id

/\* CUSTOMER NAME AND SUBTOTAL \*/

SELECT c.cus\_fname, c.cus\_lname,sum(s.pro\_price) AS SubTotal

FROM t\_customer c

INNER JOIN t\_pos\_sales s ON c.cus\_id=s.cus\_id

INNER JOIN t\_product pro ON pro.pro\_id=s.pro\_id

WHERE s.pos\_paid=0

AND s.cus\_id = @cus\_id

GROUP BY c.cus\_id,c.cus\_fname, c.cus\_lname

END

Select c.cus\_fname, c.cus\_lname,sum(s.pro\_price) As SubTotal

from t\_customer c

inner join t\_pos\_sales s on c.cus\_id=s.cus\_id

inner join t\_product pro on pro.pro\_id=s.pro\_id

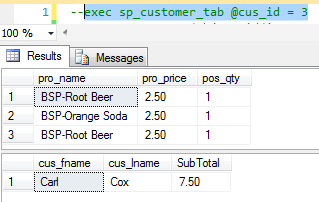
where s.pos\_paid=0

and s.cus\_id = @cus\_id

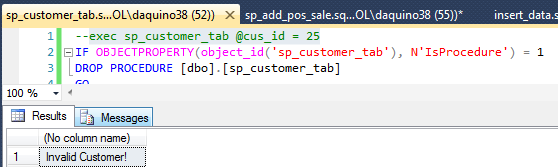
group by c.cus\_id,c.cus\_fname, c.cus\_lname

End

This screen capture demonstrates an execution of stored procedure for an existing customer.



This screen capture demonstrates the validation for an existing customer.



**Requirement 22** - The system must be able to predict when to order inventory based on sales and current inventory levels.

This query uses 30 days of average sales to determine how many days of a product remain, given the current level of inventory.

Use GIBINO

IF OBJECT\_ID('dbo.t\_tmp\_sales\_30', 'U') IS NOT NULL

drop table dbo.t\_tmp\_sales\_30

go

IF OBJECT\_ID('dbo.t\_tmp\_inv\_remain', 'U') IS NOT NULL

drop table dbo.t\_tmp\_inv\_remain

go

--create temporary table showing average sales for past thirty days

select a.pro\_id, sum(a.acct\_qty)/30 as dailysold into t\_tmp\_sales\_30

from t\_acct\_sales a

where a.acct\_datetime >= dateadd (day,-10000, getdate() )

group by a.pro\_id

go

--create temporary table showing amount on hand, given current inventory

select p.pro\_id, p.pro\_name, p.pro\_instock/nullif(s.dailysold,0) as invdays, s.dailysold, p.pro\_instock into t\_tmp\_inv\_remain

from t\_product p left join t\_tmp\_sales\_30 s

on p.pro\_id=s.pro\_id

go

select inv\_message =

case

when inv.pro\_instock = 0 then inv.pro\_name+ ' - No' +inv.pro\_name+ ' in stock. The average daily sales are '+cast(dailysold as varchar)+' units.'

when inv.invdays is NULL then inv.pro\_name+ ' - There is not enough sales information to determine amount of ' + cast(inv.pro\_name as varchar) + ' in stock. There are currently ' + cast(inv.pro\_instock as varchar) + ' in stock.'

else inv.pro\_name+ '- There are '+cast(inv.invdays as varchar)+' days (estimated) of'+inv.pro\_name+ ' product remaining.'

end

from

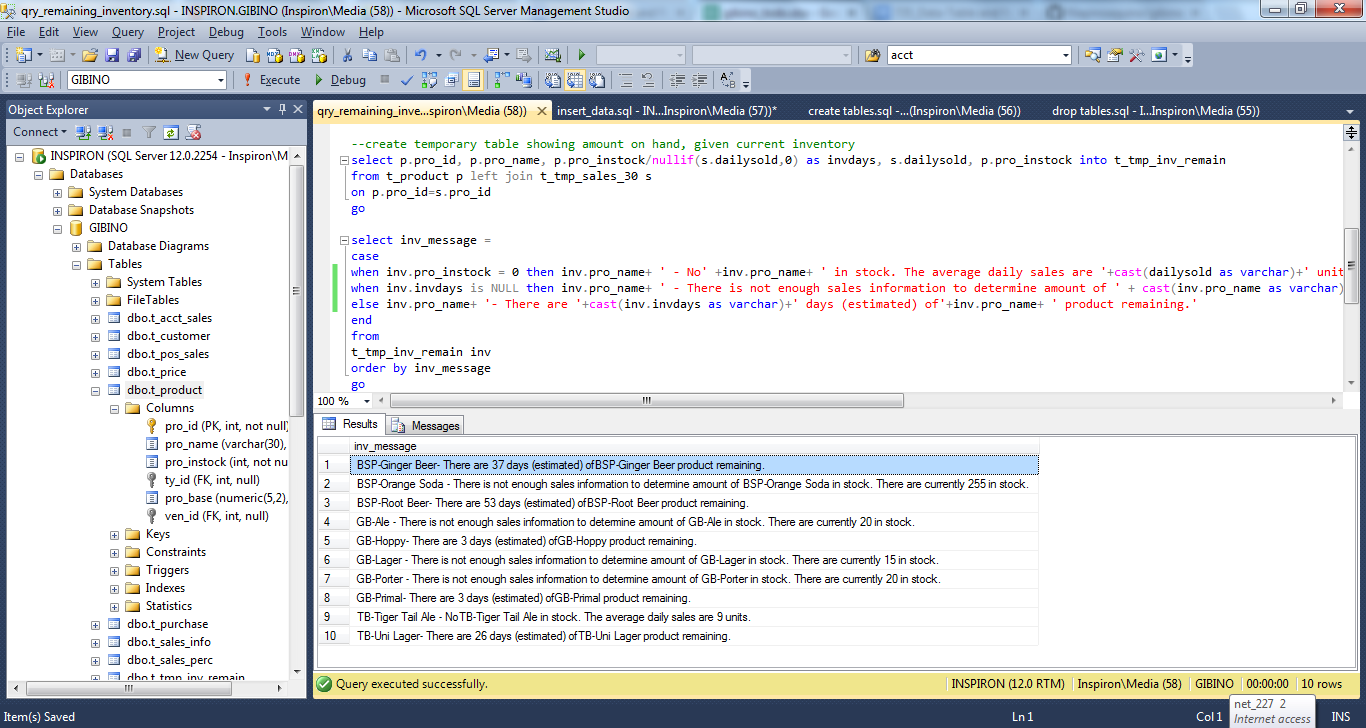
t\_tmp\_inv\_remain inv

order by inv\_message

go

The following output shows the inventory messages which appear when the query is run.

* When a beverage has a 30-day sales history and product in-stock, a message with the number of remaining days of inventory appears.
* When a beverage does not have 30-day sales history but does have product in-stock, a message indicates that there is not enough sales history to predict days-on-hand. The message also displays the number of units in stock.
* When a beverage is not in stock, the messages indicates that the beverage is not in stock, and the message also includes the average daily sales for that beverage.



**Requirement 30** – The system shall keep historical data.

The following code will create a stored procedure that will be run prior to purging the pos table to populate a permanent table which will keep historical information.

/\* CHECK IF PROCEDURE EXISTS, DROP AND RECREATE IT \*/

IF OBJECTPROPERTY(object\_id('dbo.sp\_update\_acct\_sales'), N'IsProcedure') = 1

DROP PROCEDURE [dbo].[sp\_update\_acct\_sales]

GO

CREATE PROCEDURE dbo.sp\_update\_acct\_sales

AS

BEGIN

BEGIN TRANSACTION

/\* INSERT DATA FROM THE POS TERMINAL INTO THE PERMANENT ACCOUNTING TABLE \*/

INSERT INTO T\_ACCT\_SALES (ACCT\_DATETIME, ACCT\_QTY, CUS\_ID, PRO\_ID, ACCT\_PRICE)

SELECT pos\_datetime, pos\_qty, cus\_id, pro\_id, pro\_price

FROM T\_POS\_SALES

WHERE POS\_PAID = 1;

/\* ROLLBACK ON ERROR \*/

IF @@error <> 0

BEGIN

ROLLBACK TRANSACTION

SELECT ' There was a problem migrating the sales information'

RETURN

END

COMMIT TRANSACTION;

SELECT \* FROM T\_ACCT\_SALES;

END

This screen capture demonstrates the successful insert of data from the pos table.

**Requirement 31** - The system must allow product inventory to be purchased, for valid products and vendors. When inventory is purchased from a vendor, the system must update the inventory levels in the product table.

This stored procedure is used to purchase inventory. It validates that the vendor and product exists. It also validates that the purchase quantity is greater than zero. It writes to the purchase table and updates the inventory level in the products table.

drop procedure sp\_purchase\_inventory

go

create procedure sp\_purchase\_inventory

(@pro\_id int, @ven\_id int, @pur\_qty int, @pur\_unt\_price numeric(5,2), @pur\_date datetime)

as

begin

--Checks that vendor and product exist in t\_product

if not exists (select \* from t\_product where ven\_id=@ven\_id and pro\_id=@pro\_id)

begin

select 'Vendor and product do not exist in the product table'

return

end

else

--Checks that quantity purchase is >0

if @pur\_qty <=0

begin

select 'A quantity greater than 0 must be purchased.'

return

end

--Inserts values into purchase table

begin transaction

insert into t\_purchase

(pro\_id, ven\_id, pur\_qty, pur\_unt\_price, pur\_date)

values

(@pro\_id, @ven\_id, @pur\_qty, @pur\_unt\_price, @pur\_date);

--Updates inventory amount in t\_product

update t\_product

set pro\_instock=pro\_instock+@pur\_qty

where pro\_id=@pro\_id

if @@error<>0

begin

rollback transaction

select 'Purchase was not completed.'

return

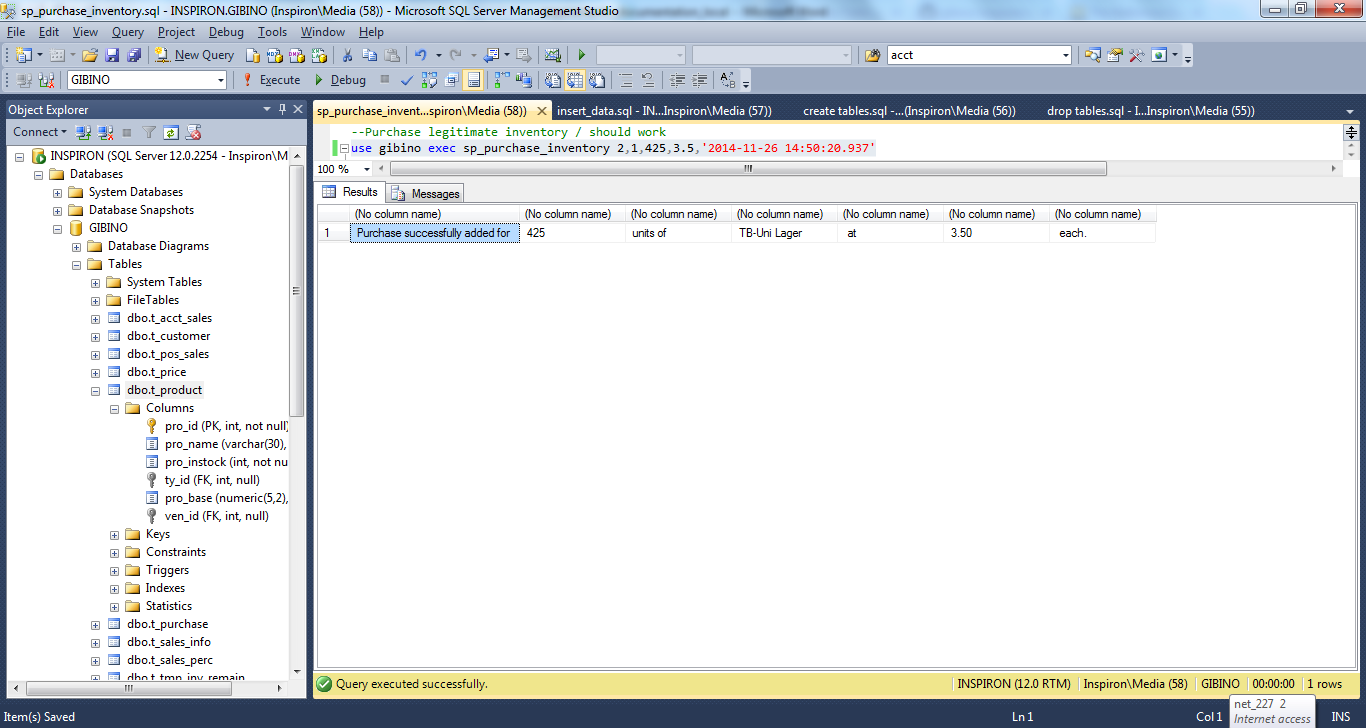
end

commit transaction

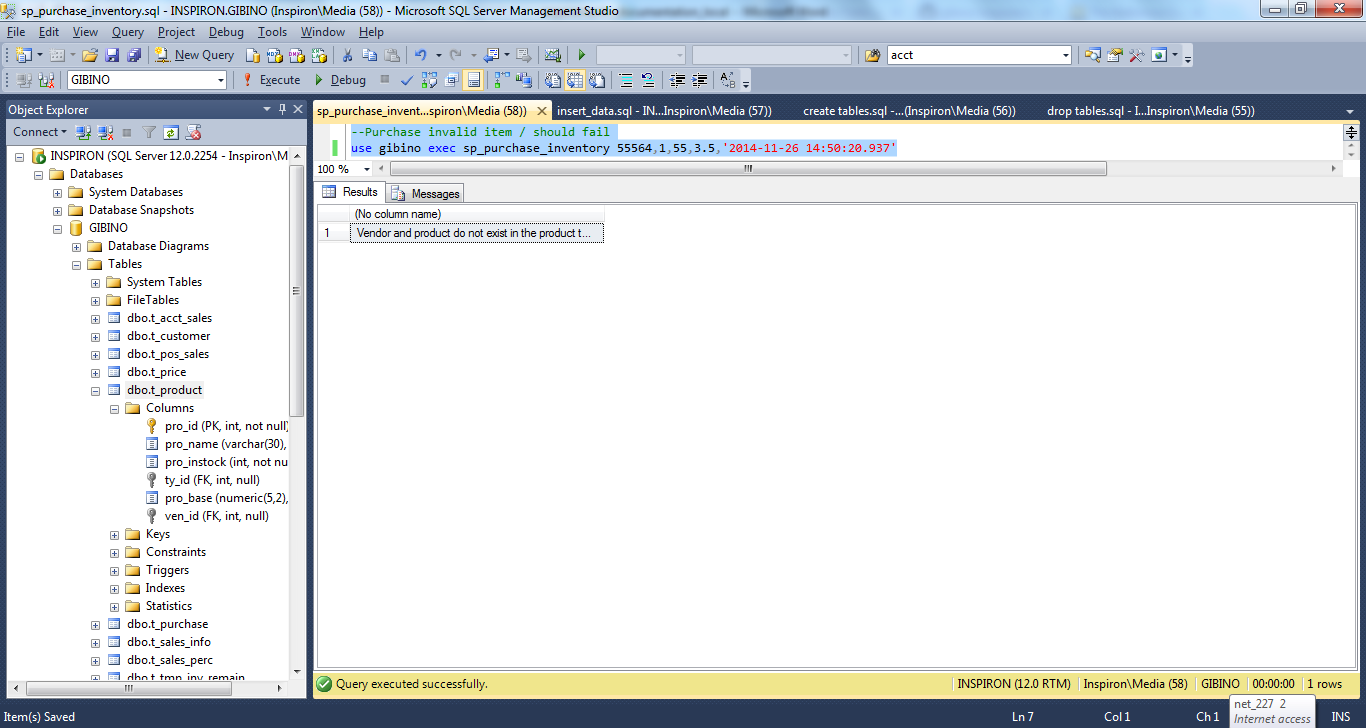
select 'Purchase successfully added for ', @pur\_qty, 'units of ', (select pro\_name from t\_product where pro\_id=@pro\_id), ' at ', @pur\_unt\_price, ' each.'

end

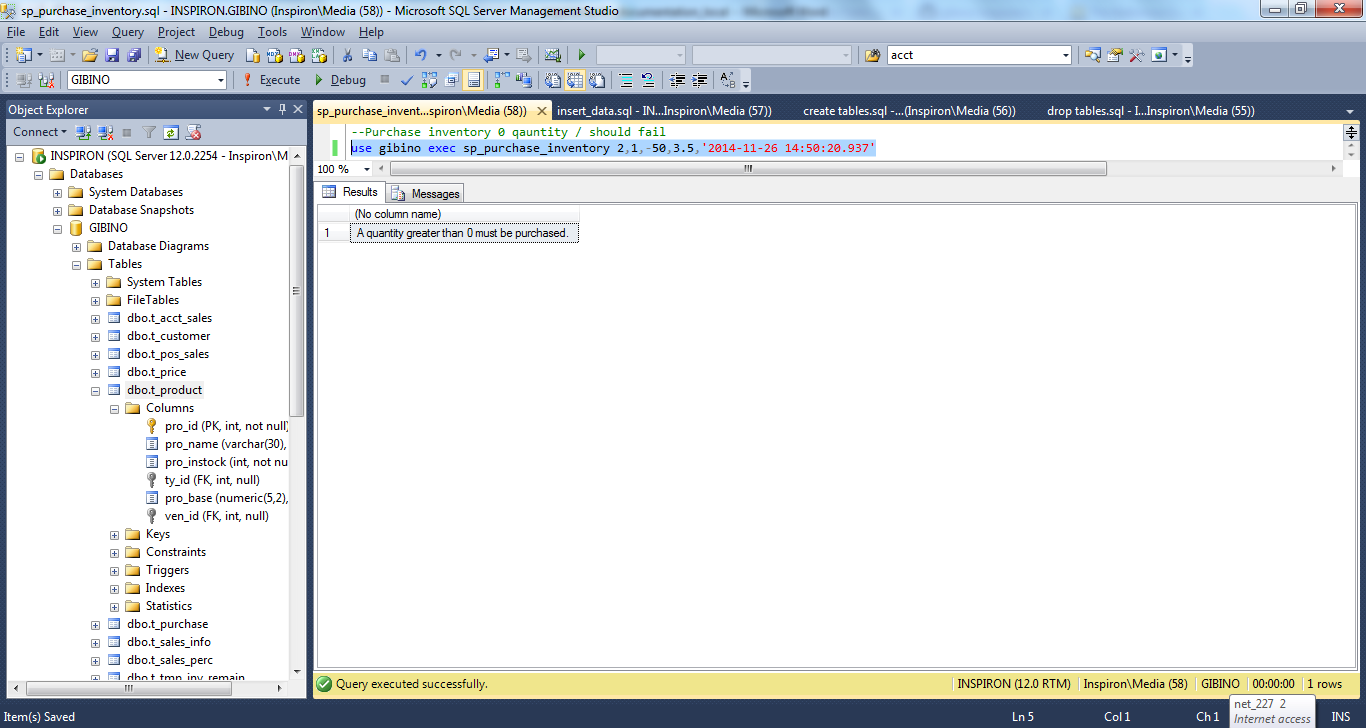
This screenshot shows a successful purchase of a valid product.



This screenshot shows a failed attempt to purchase an invalid product.



This screenshot shows a failed attempt to purchase an invalid amount (<=0) of a product.



**Requirement 32** – The system must be able to reset product prices to their original price.

The following code will create a stored procedure to reset product prices.

/\* CHECK IF PROCEDURE EXISTS, DROP AND RECREATE IT \*/

IF OBJECTPROPERTY(object\_id('dbo.sp\_reset\_price'), N'IsProcedure') = 1

DROP PROCEDURE [dbo].[sp\_reset\_price]

GO

CREATE PROCEDURE dbo.sp\_reset\_price

AS

BEGIN

BEGIN TRANSACTION

/\* RESET PRICES BACK TO THEIR ORIGINAL PRICES \*/

UPDATE pri

SET pri.pro\_price = pro.pro\_base

FROM t\_price pri INNER JOIN t\_product pro

ON pri.pro\_id = pro.pro\_id;

IF @@error <> 0

BEGIN

ROLLBACK TRANSACTION

SELECT ' Sale was NOT completed'

RETURN

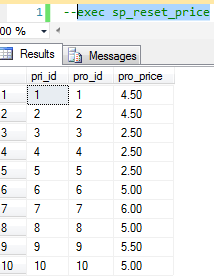
END

COMMIT TRANSACTION

SELECT \* FROM T\_PRICE;

END

This screen capture demonstrates the successful reset of product prices.



**Requirement 33** - The system must be able to add new product types without duplicates.

This stored procedure inserts a new type (i.e., a volume and category of beverage). It validates that the type does not already exist in the system.

drop procedure sp\_add\_pro\_type

go

create procedure sp\_add\_pro\_type

(@ty\_description varchar(50), @ty\_restricted int)

as

begin

--Checks for duplicate product type

if exists (select \* from t\_type where upper(replace(ty\_description, ' ', ''))=upper(replace(@ty\_description, ' ', '')))

begin

select UPPER(@ty\_description), 'already exists as a product type in the system.'

return

end

--Inserts vendor information into t\_vendor table

begin transaction

insert into t\_type

(ty\_description, ty\_restricted)

values

(@ty\_description, @ty\_restricted)

if @@error<>0

begin

rollback transaction

select @ty\_description, ' not added to the system.'

return

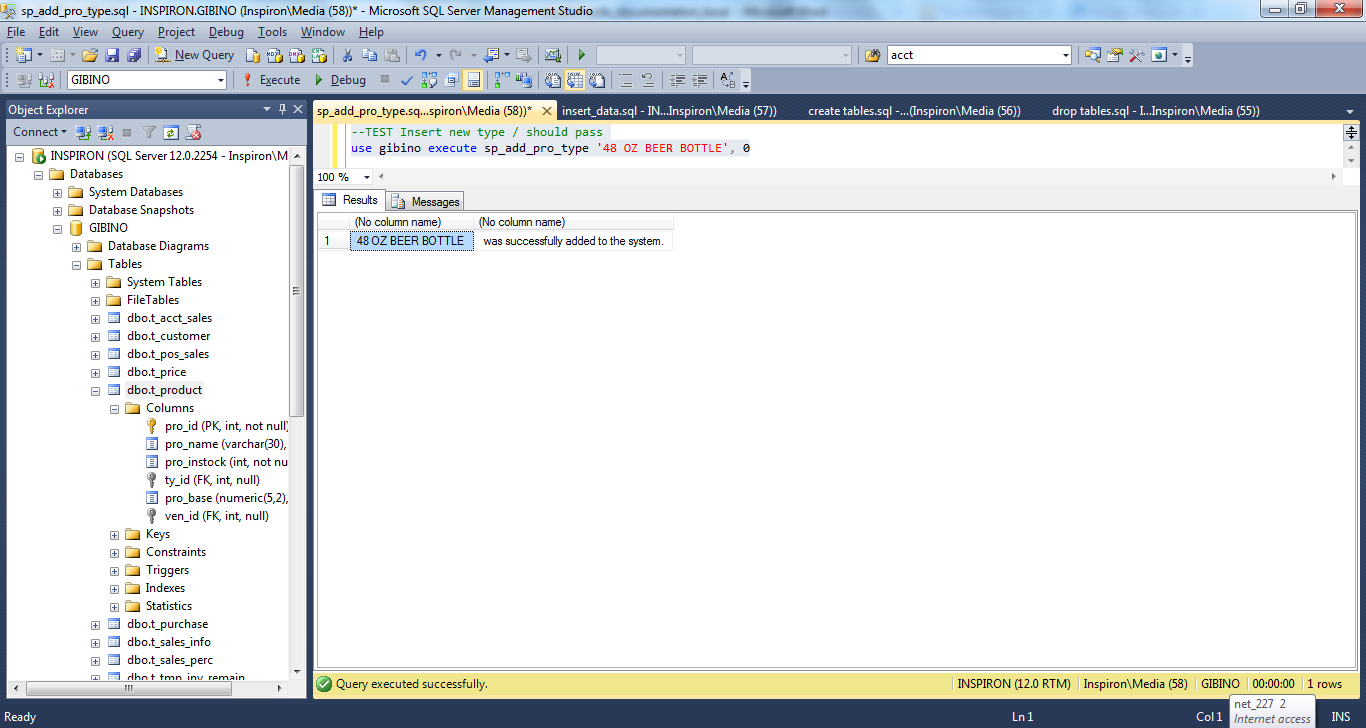
end

commit transaction

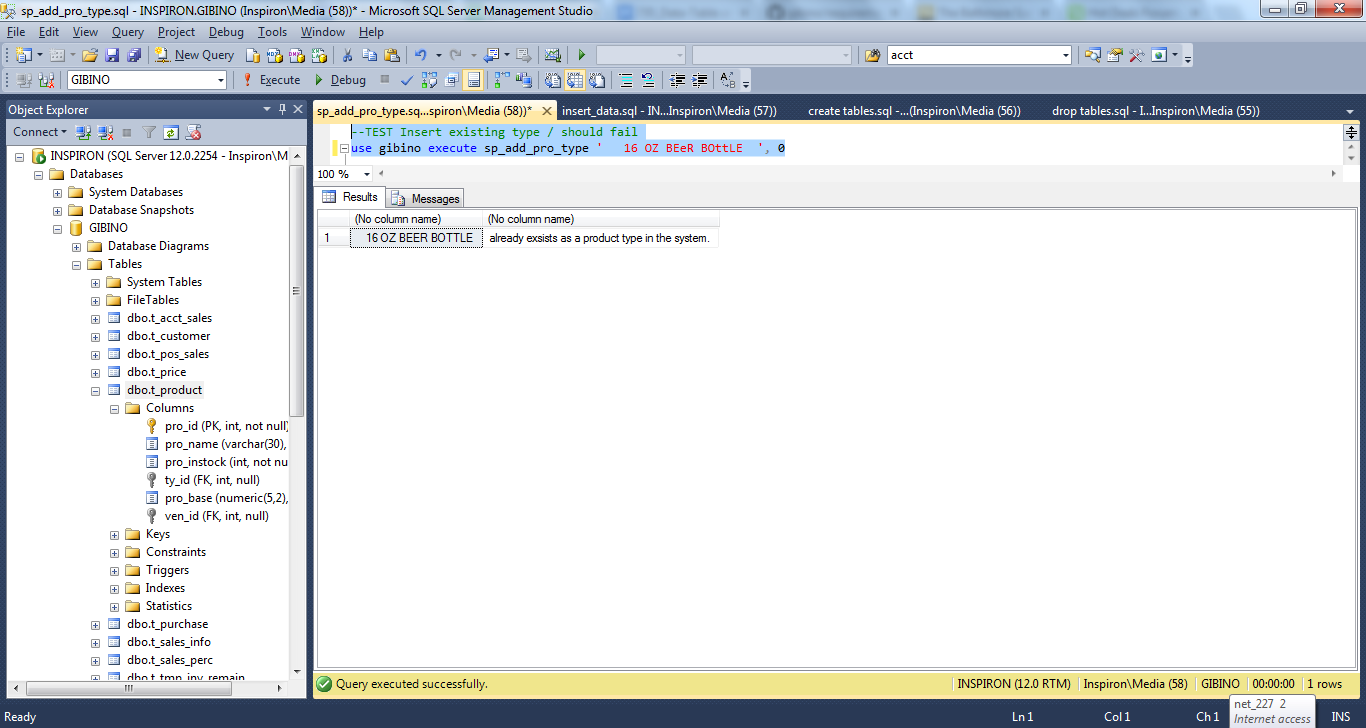
select @ty\_description, ' was successfully added to the system.'

end

This screenshot shows a successful insertion of a new type of product.



This screenshots shows a failed attempt to insert a type of product which already exists in the system.



**Requirement 35** - The system must be able to insert new customers without duplicates.

This stored procedure is used to insert new customers. It validates that the system does not already contain a customer with the same first name, last name, and date of birth.

drop procedure sp\_add\_customer

go

create procedure sp\_add\_customer

(@cus\_dob datetime, @cus\_street1 varchar(30), @cus\_street2 varchar(30), @cus\_city varchar(30), @cus\_state char(2), @cus\_zip numeric(5,0),

@cus\_email varchar(50), @cus\_fname varchar(25), @cus\_lname varchar(30), @cus\_mi varchar(1), @cus\_suffix varchar(5))

as

begin

--Checks for duplicate customer names with the same date of birth

if exists (

select \* from t\_customer where upper(replace(cus\_fname, ' ', '' ))=upper(replace(@cus\_fname, ' ', '' ))

and upper(replace(cus\_lname, ' ', '' ))=upper(replace(@cus\_lname, ' ', '' ))

and cast(cus\_dob as date)=cast(@cus\_dob as date)

)

begin

select @cus\_fname, @cus\_lname, 'already exists in this system as a customer.'

return

end

--Inserts vendor information into t\_customer table

begin transaction

insert into t\_customer

(cus\_dob, cus\_street1, cus\_street2, cus\_city, cus\_state, cus\_zip,

cus\_email, cus\_fname, cus\_lname, cus\_mi, cus\_suffix)

values

(@cus\_dob, @cus\_street1, @cus\_street2, @cus\_city, @cus\_state, @cus\_zip,

@cus\_email, @cus\_fname, @cus\_lname, @cus\_mi, @cus\_suffix)

if @@error<>0

begin

rollback transaction

select 'Customer, ', @cus\_fname, @cus\_lname, ' not added.'

return

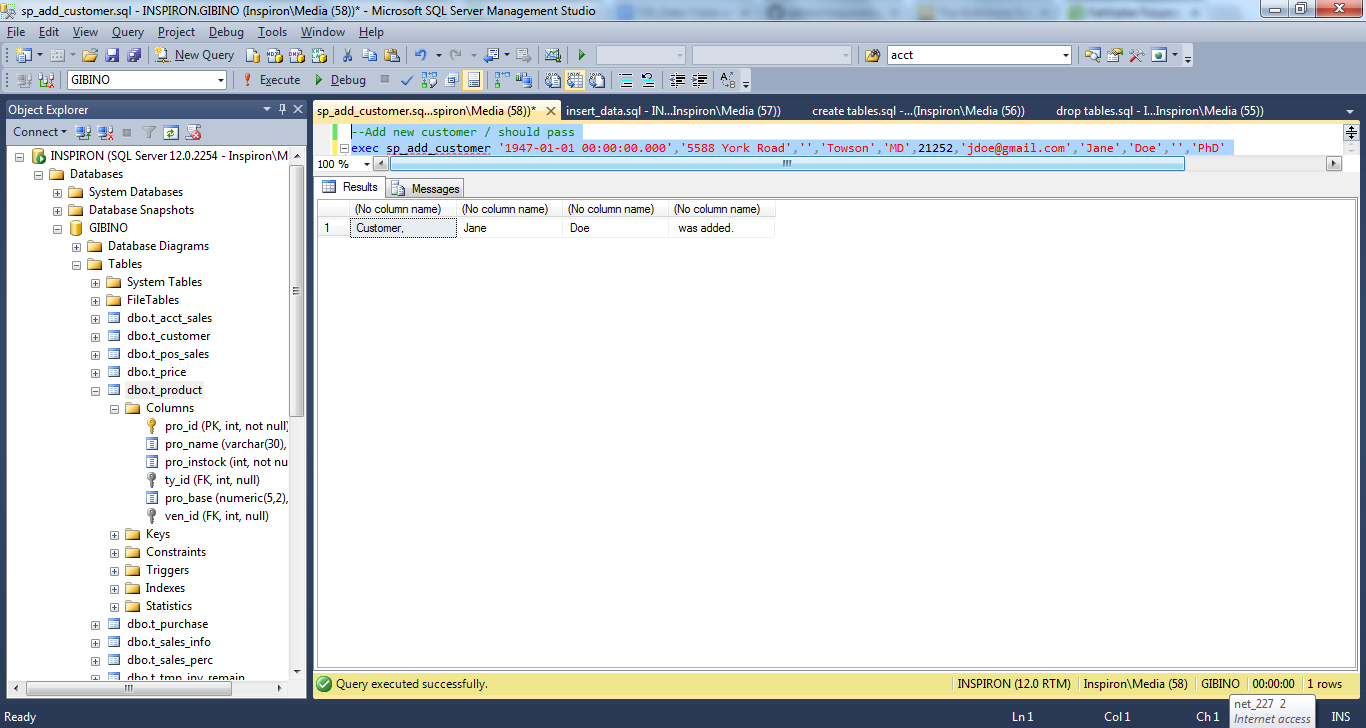
end

commit transaction

select 'Customer, ', @cus\_fname, @cus\_lname, ' was added.'

end

This screenshot shows the successful insertion of a new customer.



This screenshots shows a failed attempt to insert a customer with the same first name, last name, and date of birth as an existing customer.