**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 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(pro\_name)=upper(@pro\_name))

begin

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

return

end

else

--NEED Checks that type is valid

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

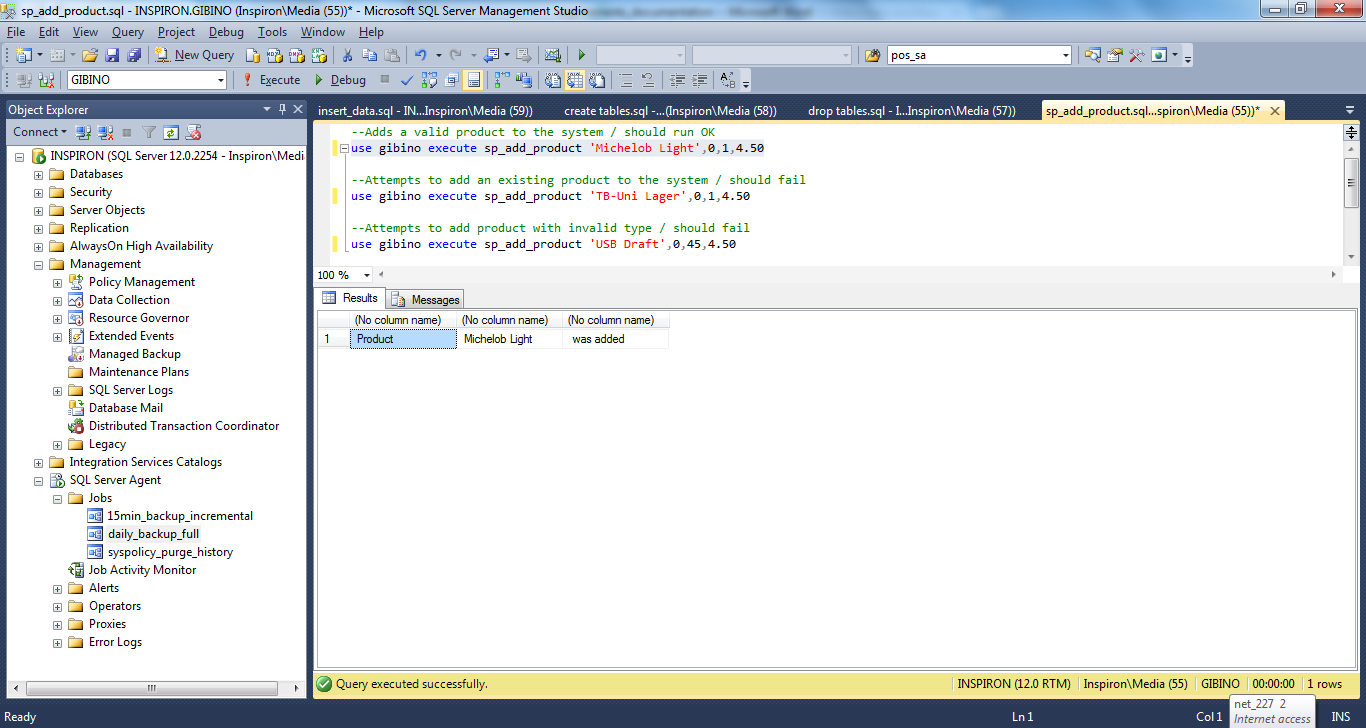
end

commit transaction

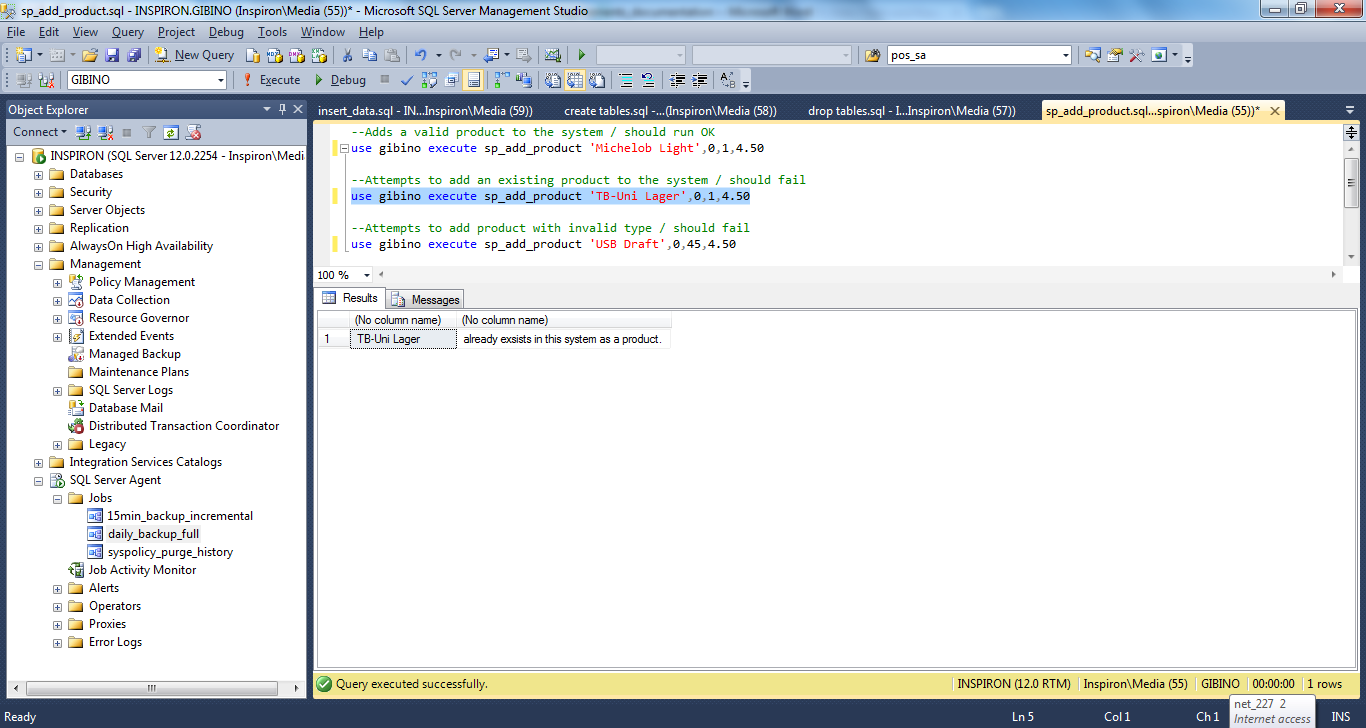
select 'Product ', @pro\_name, ' was added'

end

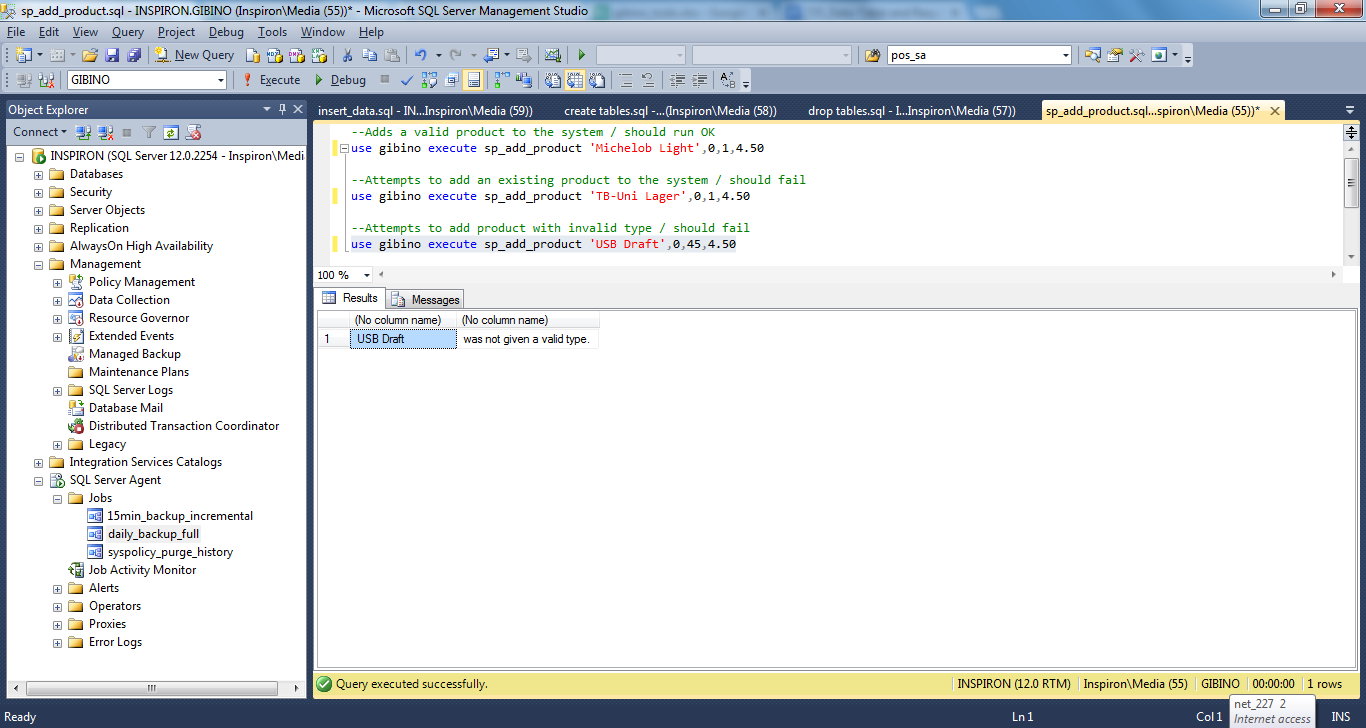
This screen shot demonstrates the insertion of a successful product.



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(ven\_name)=upper(@ven\_name))

begin

select @ven\_name, 'already exsists 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

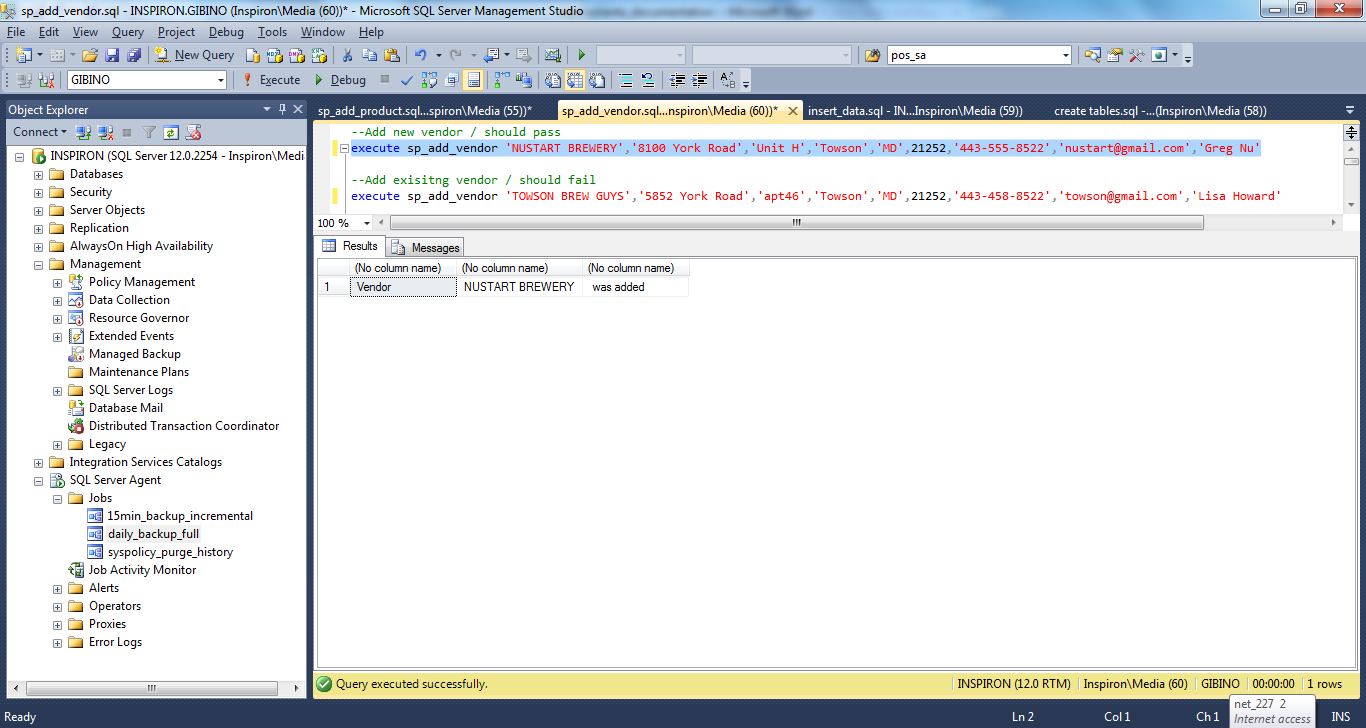
end

commit transaction

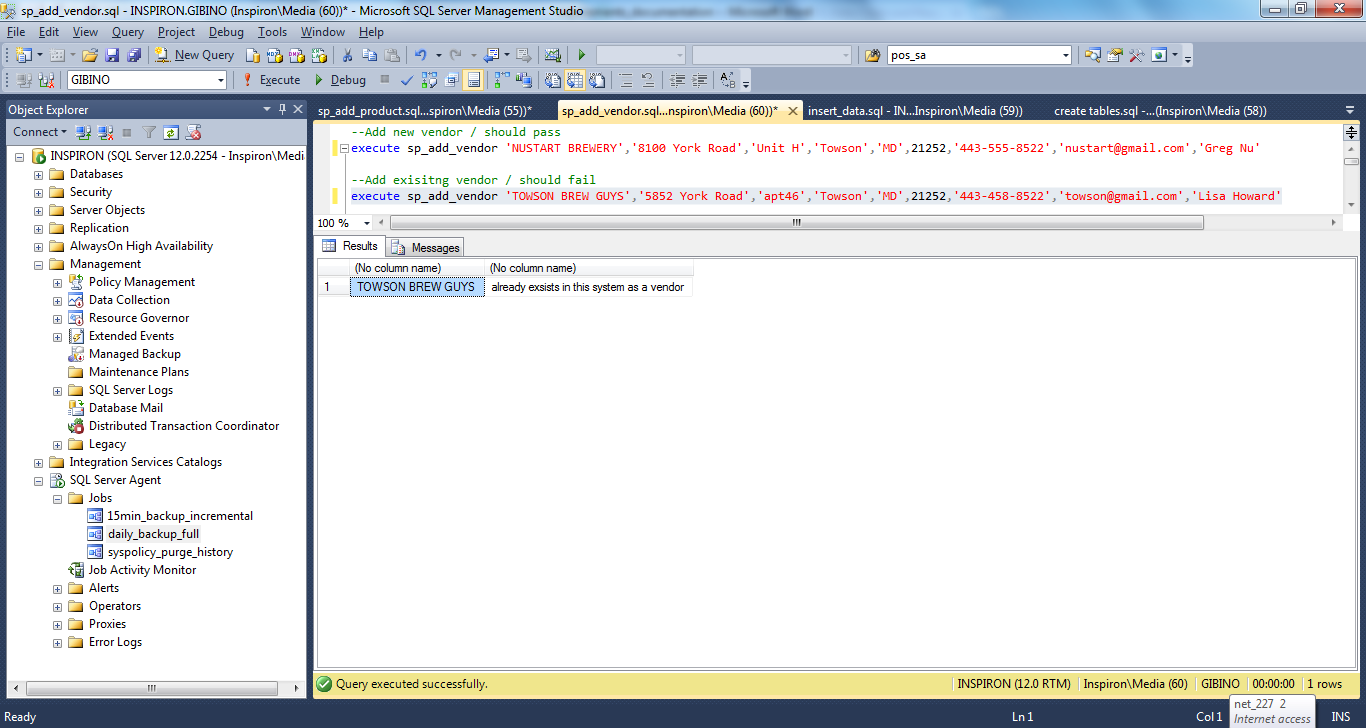
select 'Vendor ', @ven\_name, ' was added'

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

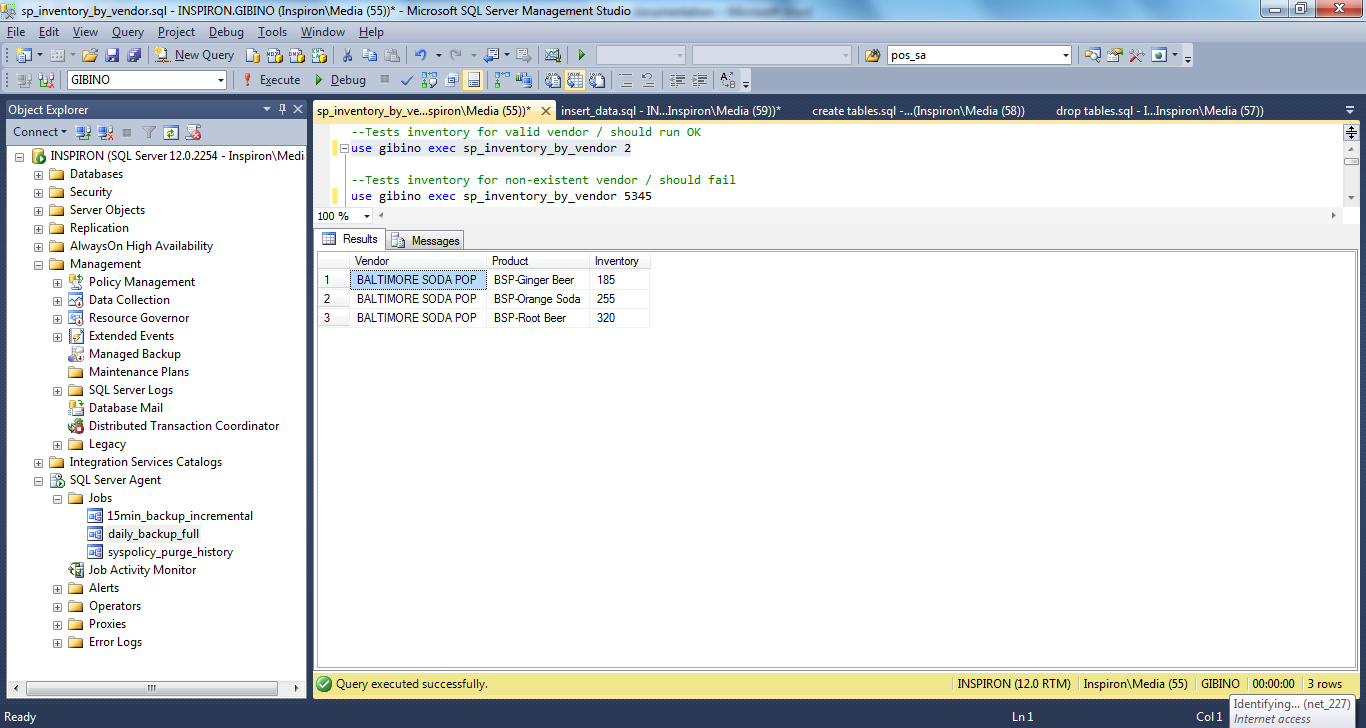
select @ven\_id, 'is not a valid vendor.'

return

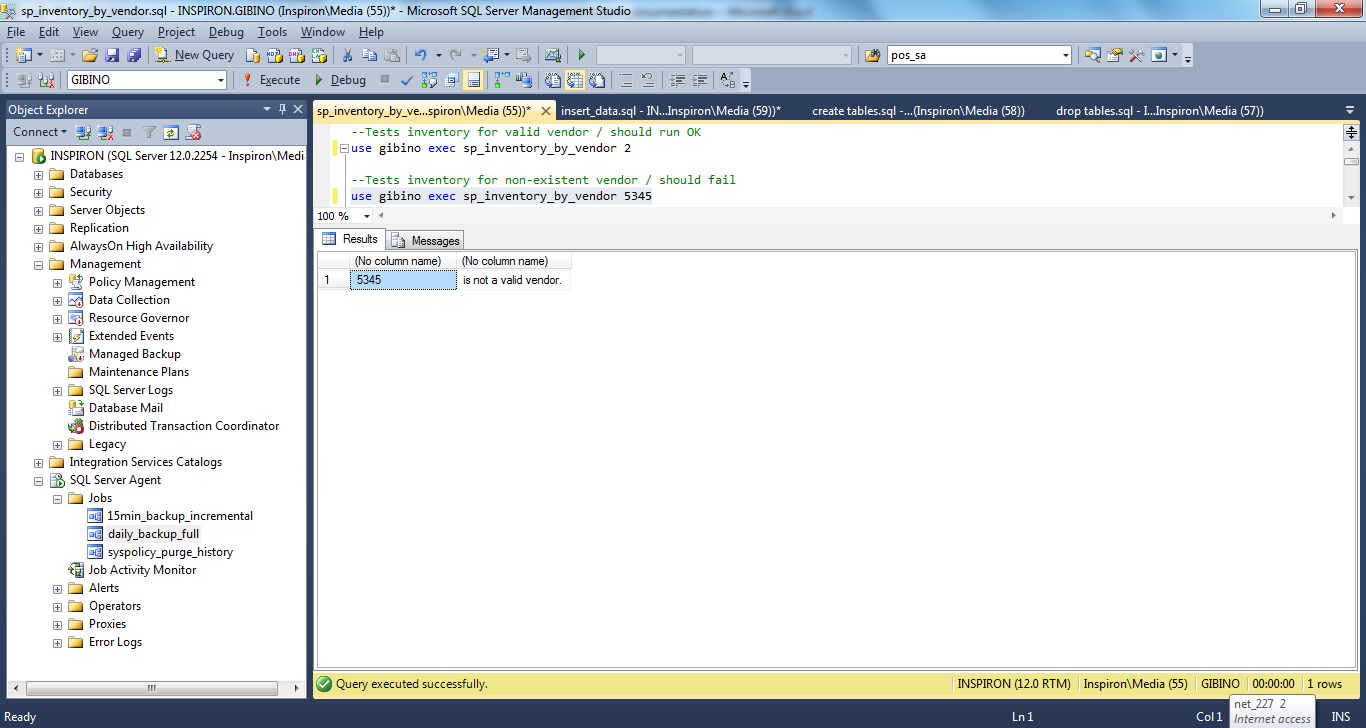
end

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 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.

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

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

);

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;

if @@error <> 0

begin

rollback transaction

select ' There was a problem creating the price report'

return

end

commit transaction

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

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

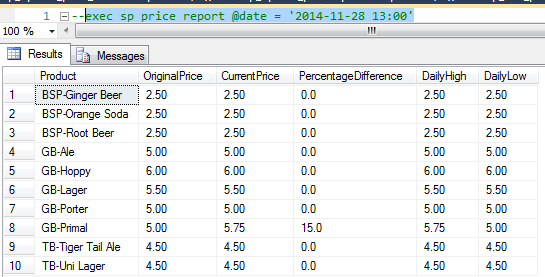
inner join t\_product pro on pri.pro\_id = pro.pro\_id

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.

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

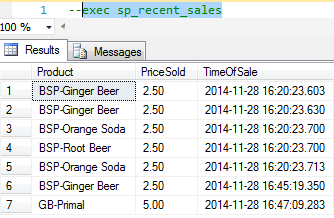
from t\_pos\_sales s

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.

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 pro\_name, max(pos\_qty) as TopSeller

from t\_pos\_sales s

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

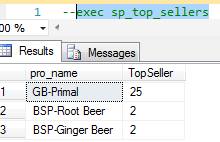
where Day(pos\_datetime) = Day(GetDate())

group by p.pro\_id, pro\_name

order by TopSeller desc;

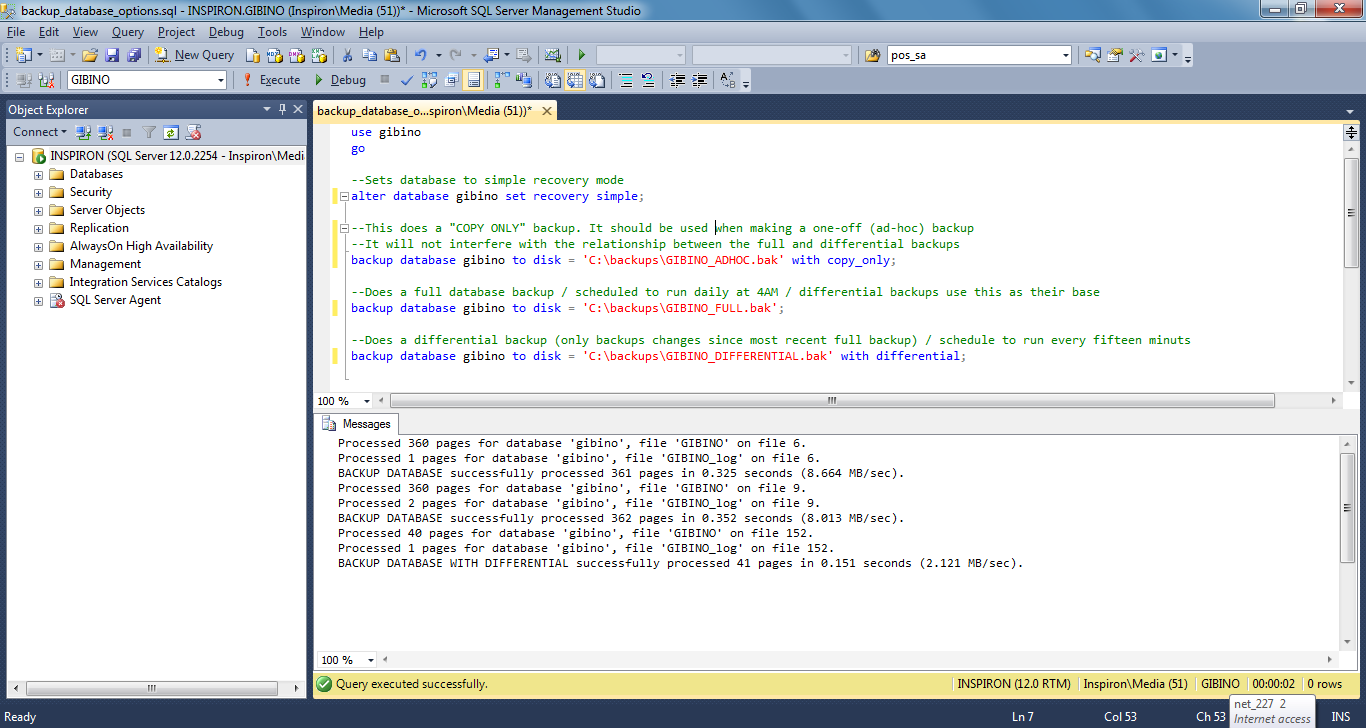
END

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

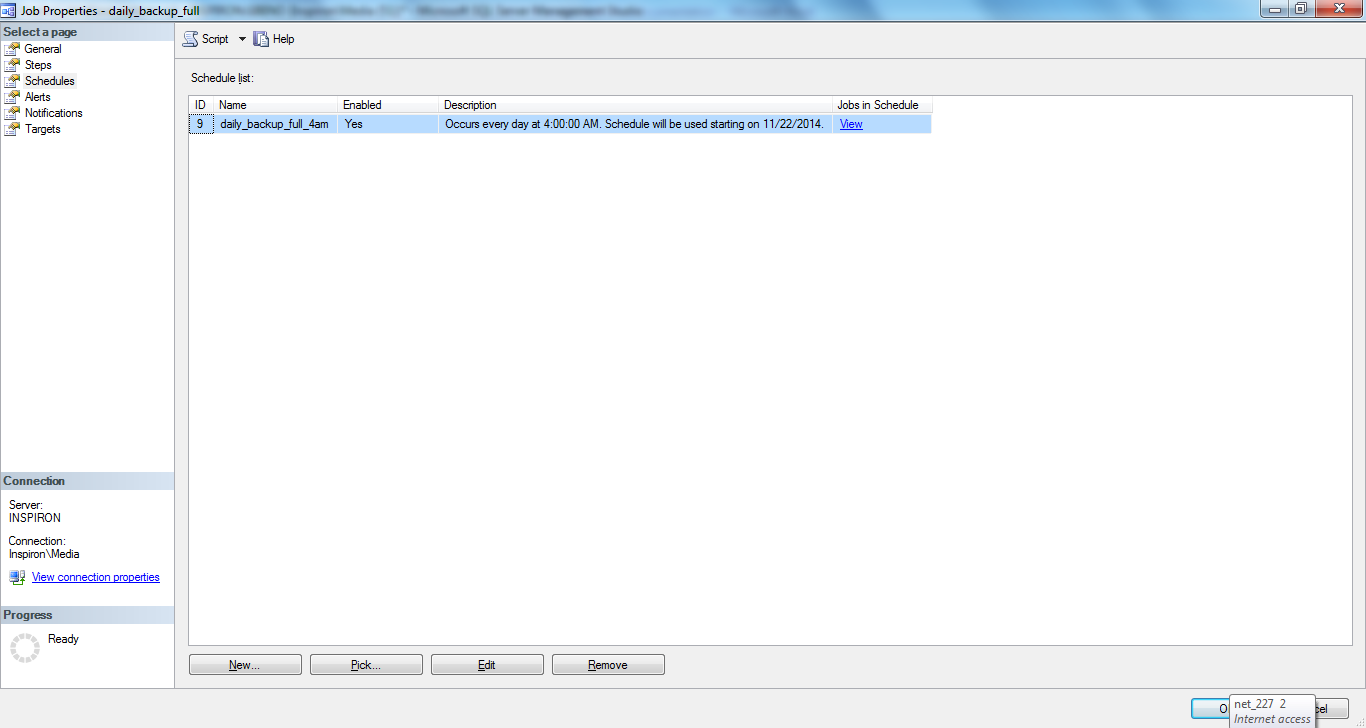


**Requirement 14** -The system must create a nightly backup, after normal business hours.

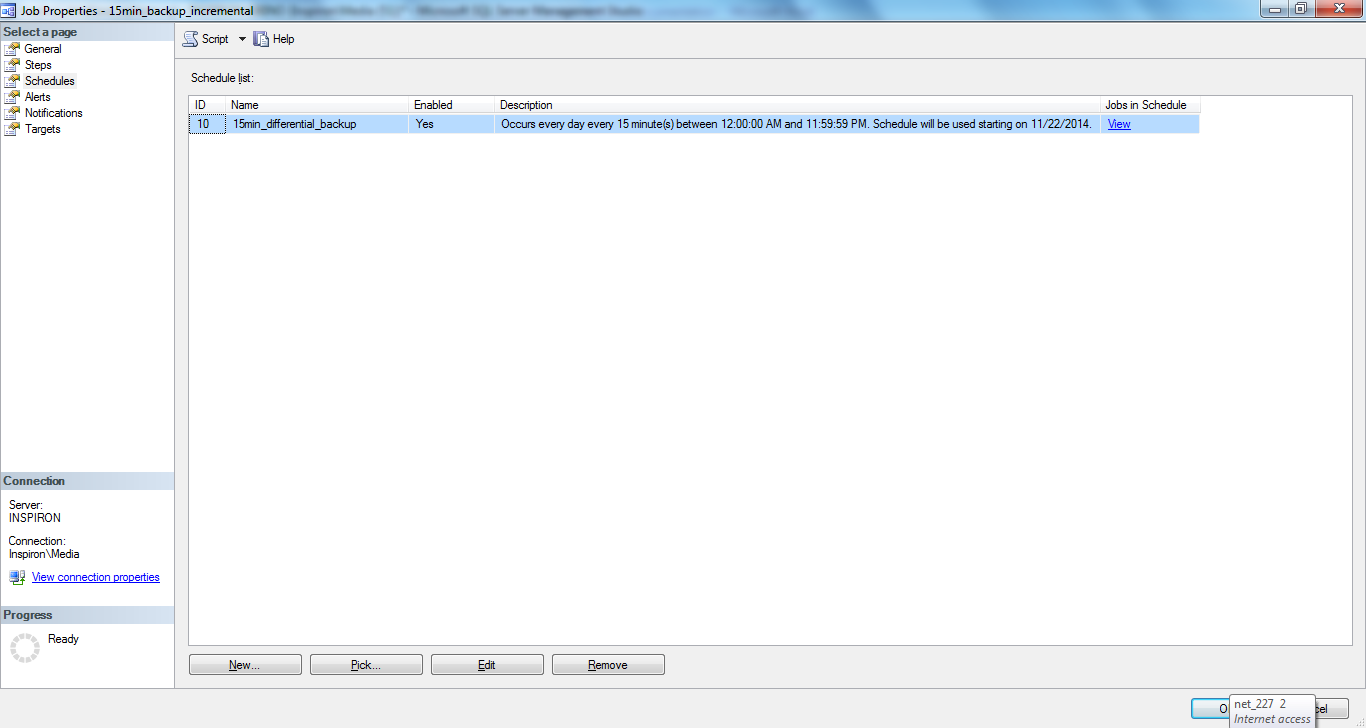
The following code demonstrates how to setup the database for ad hoc, full, and differential backups. It also shows how to put the database into simple recovery mode.



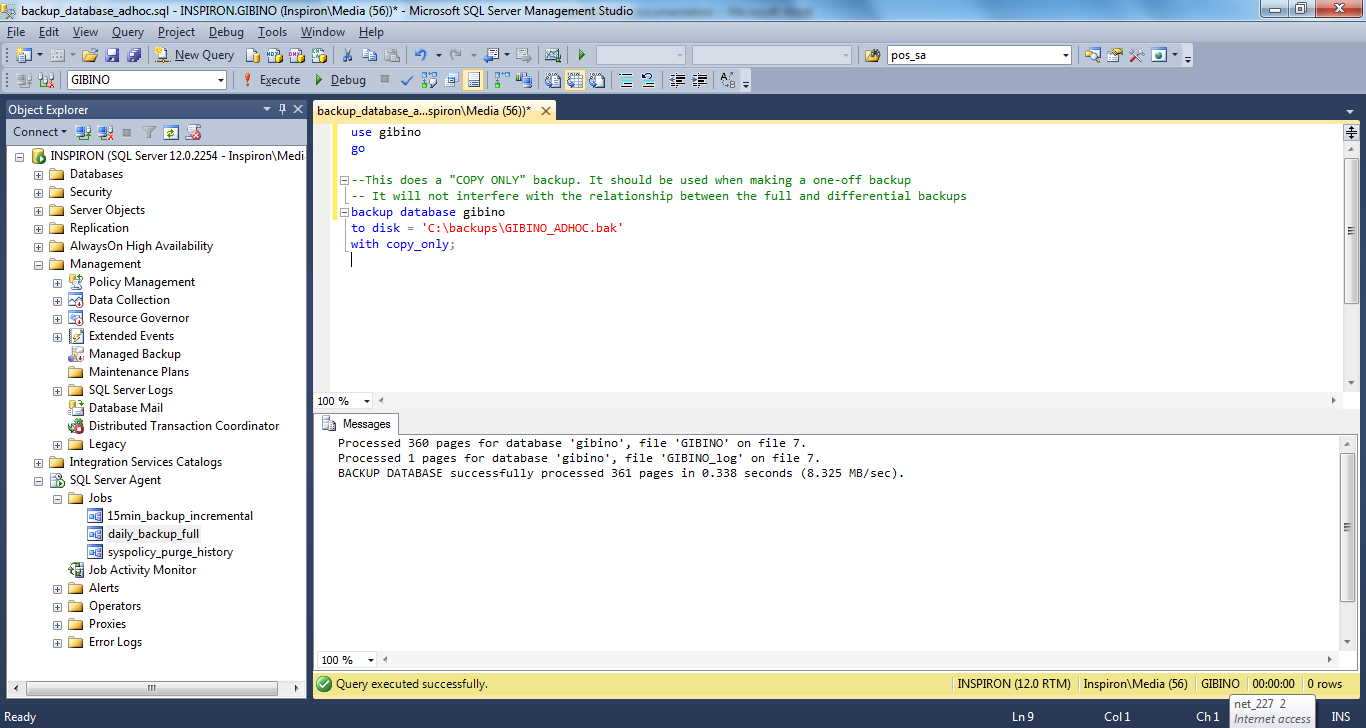
This screen capture shows that the full backup is a scheduled job, which runs at 4:00 AM every day.



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.

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

@start\_time datetime,

@end\_time datetime

AS

BEGIN

BEGIN TRANSACTION;

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

);

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

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

);

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

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

);

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 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;

if @@error <> 0

begin

rollback transaction

select ' Price update was unsuccessful'

return

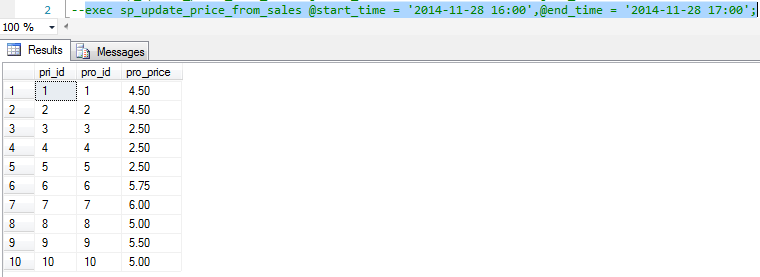
end

commit transaction;

SELECT \* FROM T\_PRICE;

END

This screen capture demonstrates successful update of product prices.



**Requirement 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(ven\_name)=upper(@ven\_name) and ven\_id <> @ven\_id)

begin

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

return

end

--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

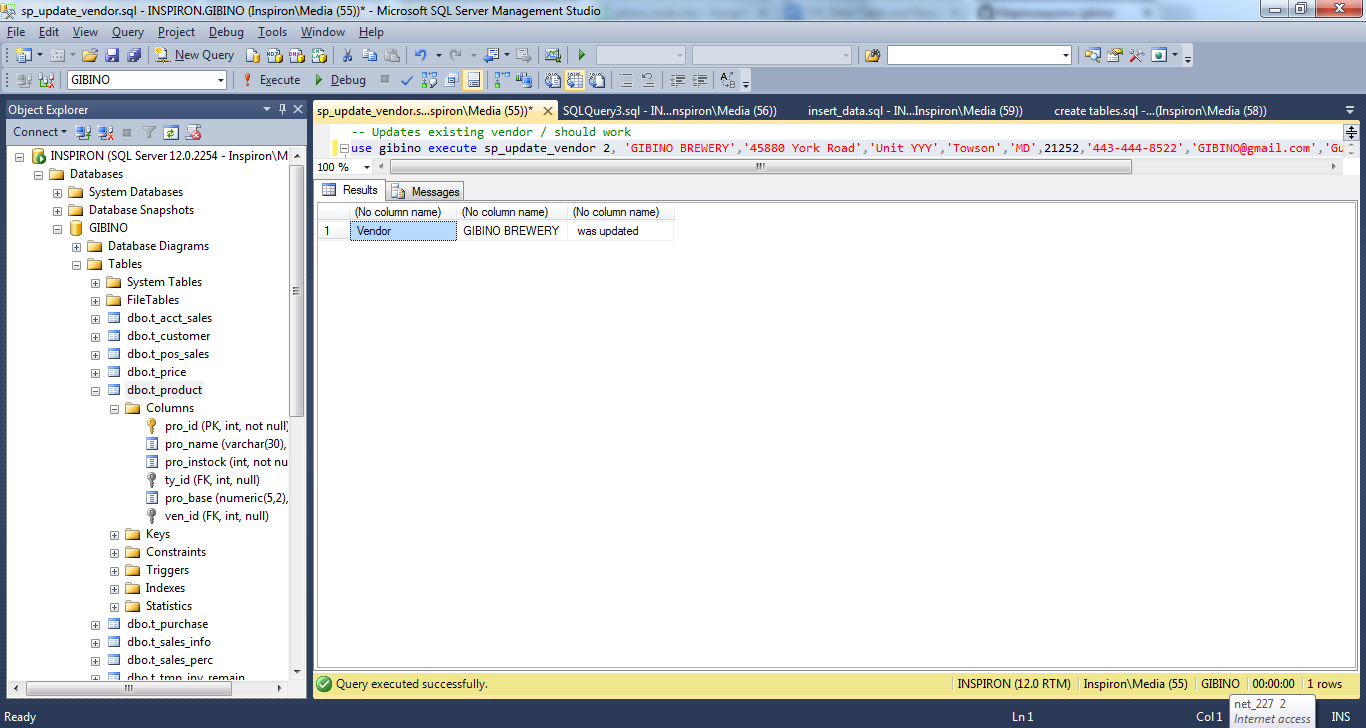
end

commit transaction

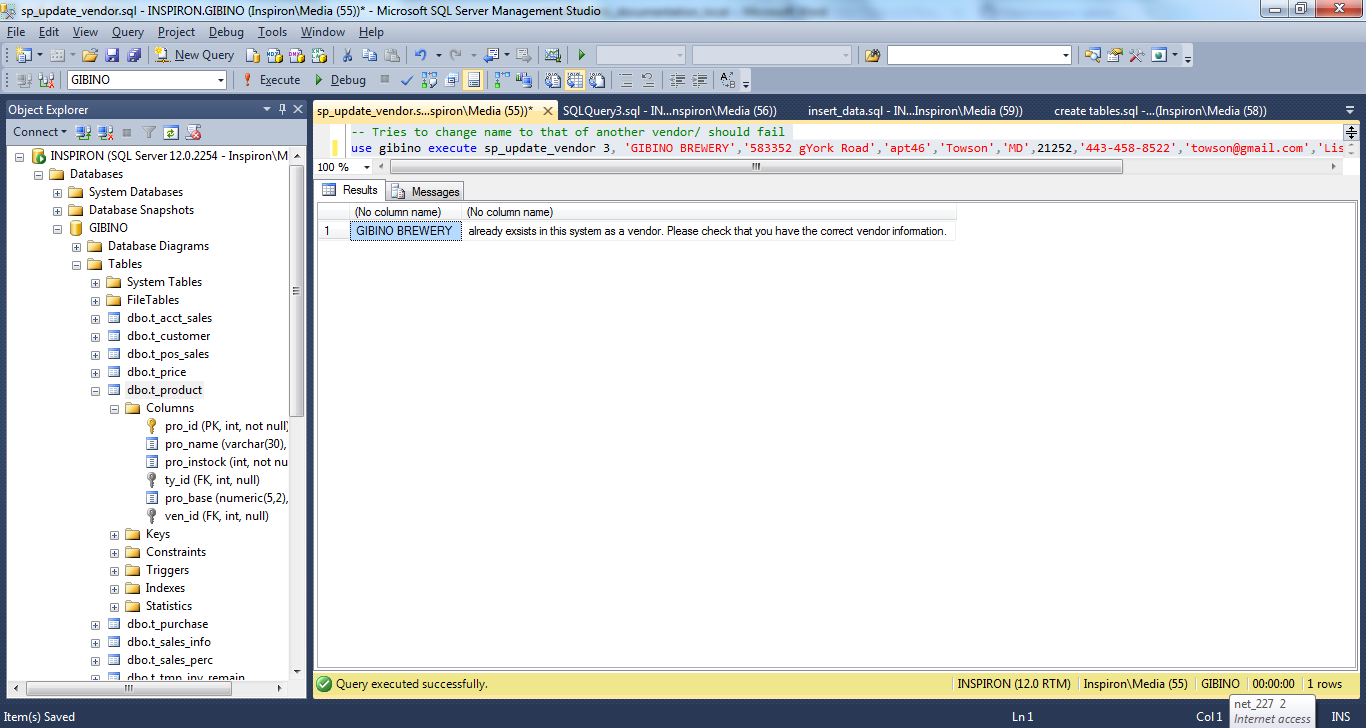
select 'Vendor ', @ven\_name, ' was updated'

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.

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

If not exists (Select \* from T\_CUSTOMER where cus\_id = @cus\_id)

begin

select 'Invalid Customer!'

return

end

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

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

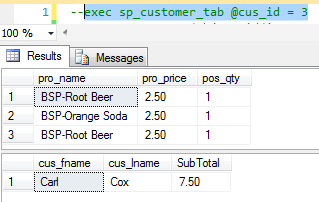
where s.pos\_paid=0

and s.cus\_id = @cus\_id

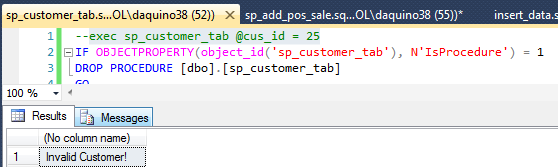
group by c.cus\_id,c.cus\_fname, c.cus\_lname

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

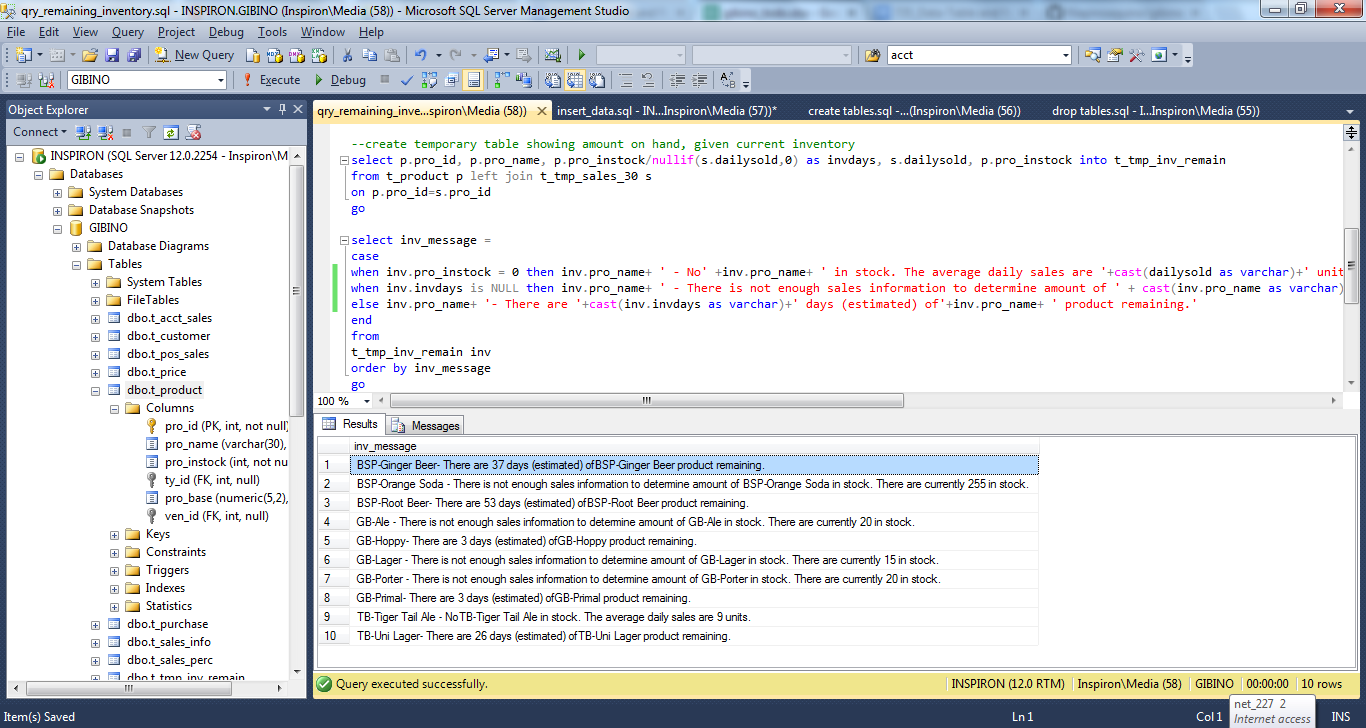
t\_tmp\_inv\_remain inv

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 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.

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

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.

