AdventureWorks Extended Order Details Script

This comprehensive SQL script (for SQL Server 2019+) sets up a sample AdventureWorks-like environment and implements the requested stored procedures, functions, views, and triggers. It includes a **setup** section (creating schemas/tables and sample data if needed), then defines the required objects. Key SQL Server features are used (default parameter values, `ISNULL`, `CONVERT` with style codes, triggers, etc.), with logic explained below and references to relevant documentation.

Setup & Sample Data

- **Database and Schemas:** Checks for the `AdventureWorks` database (creates it if missing) and creates schemas `Sales`, `Production`, `Purchasing`.
- **Tables:** Creates minimal tables `Sales.Customers`, `Sales.Orders`, `Sales.OrderDetails`, `Production.Product`, `Production.Supplier`, `Production.Category`. Fields include customer name, order dates, product inventory (with `UnitsInStock` and `ReorderLevel`), etc. Foreign keys enforce relationships.
- **Sample Rows:** Inserts a few sample customers, suppliers, categories, products (with stock levels), and one order with details.

This ensures the environment works even if a full AdventureWorks DB is not installed. In practice, you could attach Microsoft's official AdventureWorks OLTP backup instead (see AdventureWorks install instructions [1]).

```
-- SETUP: Database & Tables
IF DB_ID('AdventureWorks') IS NULL
BEGIN
    CREATE DATABASE AdventureWorks;
END;
GO
USE AdventureWorks;
GO

-- Create schemas if not exist
IF NOT EXISTS (SELECT * FROM sys.schemas WHERE name = 'Sales') EXEC('CREATE
SCHEMA Sales');
IF NOT EXISTS (SELECT * FROM sys.schemas WHERE name = 'Production')
EXEC('CREATE SCHEMA Production');
IF NOT EXISTS (SELECT * FROM sys.schemas WHERE name = 'Purchasing')
EXEC('CREATE SCHEMA Purchasing');

-- Drop existing tables (idempotency)
IF OBJECT_ID('Sales.OrderDetails') IS NOT NULL DROP TABLE Sales.OrderDetails;
```

```
IF OBJECT_ID('Sales.Orders') IS NOT NULL DROP TABLE Sales.Orders;
IF OBJECT_ID('Sales.Customers') IS NOT NULL DROP TABLE Sales.Customers;
IF OBJECT_ID('Production.Product') IS NOT NULL DROP TABLE Production.Product;
IF OBJECT_ID('Production.Supplier') IS NOT NULL DROP TABLE Production.Supplier;
IF OBJECT_ID('Production.Category') IS NOT NULL DROP TABLE Production.Category;

-- Create tables
CREATE TABLE Sales.Customers (
CustomerID INT PRIMARY KEY IDENTITY(1,1),
CompanyName NVARCHAR(100) NOT NULL
);
CREATE TABLE Sales.Orders (
OrderID INT PRIMARY KEY IDENTITY(1000,1),
    CustomerID INT NOT NULL REFERENCES Sales.Customers(CustomerID), OrderDate
DATETIME NOT NULL );
CREATE TABLE Production.Supplier (
SupplierID INT PRIMARY KEY IDENTITY(1,1),
    CompanyName NVARCHAR(100) NOT NULL );
CREATE TABLE Production.Category (
CategoryID INT PRIMARY KEY IDENTITY(1,1),
CategoryName NVARCHAR(100) NOT NULL
);
CREATE TABLE Production.Product (
ProductID INT PRIMARY KEY IDENTITY(1,1),
Name NVARCHAR(100) NOT NULL,
SupplierID INT NULL REFERENCES Production.Supplier(SupplierID),
CategoryID INT NULL REFERENCES Production.Category(CategoryID),
ListPrice MONEY NOT NULL,
UnitsInStock INT NOT NULL,
ReorderLevel INT NOT NULL DEFAULT 0,
Discontinued BIT NOT NULL DEFAULT 0
);
CREATE TABLE Sales.OrderDetails (
OrderID INT NOT NULL REFERENCES Sales.Orders(OrderID),
ProductID INT NOT NULL REFERENCES Production.Product(ProductID),
UnitPrice MONEY NOT NULL,
Quantity INT NOT NULL,
Discount DECIMAL(5,2) NOT NULL DEFAULT 0,
CONSTRAINT PK_OrderDetails PRIMARY KEY (OrderID,ProductID)
);

-- Insert sample data
INSERT INTO Sales.Customers (CompanyName) VALUES
('Acme Corp'), ('Global Industries'), ('Contoso Ltd.');
INSERT INTO Production.Supplier (CompanyName) VALUES ('Supplier A'), ('Supplier
B');
INSERT INTO Production.Category (CategoryName) VALUES ('Electronics'),
('Clothing');
INSERT INTO Production.Product (Name, SupplierID, CategoryID, ListPrice,
```

```
UnitsInStock, ReorderLevel, Discontinued) VALUES
    ('Laptop', 1, 1, 1200.00, 50, 10, 0),
    ('Smartphone', 2, 1, 800.00, 100, 20, 0),
    ('Jeans', 2, 2, 60.00, 200, 50, 0);

INSERT INTO Sales.Orders (CustomerID, OrderDate) VALUES (1, GETDATE());
INSERT INTO Sales.OrderDetails (OrderID, ProductID, UnitPrice, Quantity,
Discount) VALUES
    (1000, 1, 1200.00, 2, 0),
    (1000, 3, 55.00, 5, 0);
```

Stored Procedures

1. **InsertOrderDetails** – Inserts a new order detail row, with logic as follows:
2. **Default parameters:** `@UnitPrice MONEY = NULL` , `@Discount DECIMAL(5,2) = 0` . (Parameters with default values are optional [2].)
3. If `@UnitPrice` is NULL, the procedure looks up the product's `ListPrice` and uses that.
4. It checks `UnitsInStock` (from `Production.Product`) to ensure enough inventory exists; if not, it prints an error and exits.
5. It performs the `INSERT` . After inserting, it checks `@@ROWCOUNT` ; if zero, prints a failure message.
6. Finally, it subtracts the sold `Quantity` from `UnitsInStock` , and if the new stock is below the product's `ReorderLevel` , it prints a warning.

```
CREATE OR ALTER PROCEDURE Sales.InsertOrderDetails
    @OrderID INT,
    @ProductID INT,
    @Quantity INT,
    @UnitPrice MONEY = NULL,
    @Discount DECIMAL(5,2) = 0
AS
BEGIN
    SET NOCOUNT ON;
    -- If UnitPrice not provided, default to product's ListPrice
    IF @UnitPrice IS NULL
        SELECT @UnitPrice = ListPrice FROM Production.Product WHERE ProductID =
@ProductID;

    -- Check product exists and stock availability
    DECLARE @StockBefore INT;
    SELECT @StockBefore = UnitsInStock FROM Production.Product WHERE ProductID
= @ProductID;
    IF @StockBefore IS NULL
    BEGIN
```

```
        PRINT 'Product ID ' + CAST(@ProductID AS VARCHAR) + ' does not exist.';
        RETURN;
    END
    IF @Quantity > @StockBefore
    BEGIN
        PRINT 'Insufficient stock for ProductID ' + CAST(@ProductID AS VARCHAR);
        RETURN;
    END

    -- Insert the order detail
    INSERT INTO Sales.OrderDetails (OrderID, ProductID, UnitPrice, Quantity,
Discount)
    VALUES (@OrderID, @ProductID, @UnitPrice, @Quantity, @Discount);

    -- Verify insertion
    IF @@ROWCOUNT = 0
    BEGIN
        PRINT 'Failed to insert OrderDetails for OrderID ' + CAST(@OrderID AS
VARCHAR) + '.';
        RETURN;
    END

    -- Update stock
    UPDATE Production.Product
    SET UnitsInStock = UnitsInStock - @Quantity
    WHERE ProductID = @ProductID;

    -- Warn if stock below reorder level
    DECLARE @StockAfter INT, @ReorderLevel INT;
    SELECT @StockAfter = UnitsInStock, @ReorderLevel = ReorderLevel
      FROM Production.Product WHERE ProductID = @ProductID;
    IF @StockAfter < @ReorderLevel
    BEGIN
        PRINT 'Warning: Stock for ProductID ' + CAST(@ProductID AS VARCHAR) +
              ' has fallen below reorder level (' + CAST(@ReorderLevel AS
VARCHAR) + ').';
    END
END;
GO
```

1. **UpdateOrderDetails**– Updates an existing detail row. Parameters: `@OrderID`, `@ProductID` (required), and optional `@UnitPrice`, `@Quantity`, `@Discount`. Any NULL parameter means “no change” to that column. This is done by `ISNULL(newValue, oldValue)` logic [3]. The procedure adjusts inventory: it computes the quantity difference and updates `UnitsInStock` accordingly (with a stock check if increasing quantity).

```
CREATE OR ALTER PROCEDURE Sales.UpdateOrderDetails
    @OrderID INT,
    @ProductID INT,
    @UnitPrice MONEY = NULL,
    @Quantity INT = NULL,
    @Discount DECIMAL(5,2) = NULL
AS
BEGIN
    SET NOCOUNT ON;
    -- Verify the detail exists
    IF NOT EXISTS (SELECT 1 FROM Sales.OrderDetails
                   WHERE OrderID = @OrderID AND ProductID = @ProductID)
    BEGIN
        PRINT 'OrderDetail (' + CAST(@OrderID AS VARCHAR) + ',' +
CAST(@ProductID AS VARCHAR) + ') not found.';
        RETURN;
    END

    -- Get current values
    DECLARE @OldUnitPrice MONEY, @OldQuantity INT, @OldDiscount DECIMAL(5,2);
    SELECT @OldUnitPrice = UnitPrice, @OldQuantity = Quantity, @OldDiscount =
Discount
      FROM Sales.OrderDetails
      WHERE OrderID = @OrderID AND ProductID = @ProductID;

    -- Retain old values for NULL parameters
    SET @UnitPrice = ISNULL(@UnitPrice, @OldUnitPrice);
    SET @Quantity  = ISNULL(@Quantity,  @OldQuantity);
    SET @Discount  = ISNULL(@Discount,  @OldDiscount);

    -- Compute change in quantity
    DECLARE @QtyDiff INT = @Quantity - @OldQuantity;

    -- If increasing quantity, check stock
    IF @QtyDiff > 0
    BEGIN
        DECLARE @Stock INT;
        SELECT @Stock = UnitsInStock FROM Production.Product WHERE ProductID =
@ProductID;
        IF @Stock < @QtyDiff
        BEGIN
            PRINT 'Insufficient stock to increase quantity for ProductID '  +
CAST(@ProductID AS VARCHAR);
            RETURN;
        END
    END
```

```
    -- Perform update
    UPDATE Sales.OrderDetails
    SET UnitPrice = @UnitPrice,
        Quantity = @Quantity,
        Discount = @Discount
    WHERE OrderID = @OrderID AND ProductID = @ProductID;

    -- Adjust stock if needed
    IF @QtyDiff <> 0
    BEGIN
        UPDATE Production.Product
        SET UnitsInStock = UnitsInStock - @QtyDiff
        WHERE ProductID = @ProductID;
    END
END;
GO
```

1. **GetOrderDetails**– Takes `@OrderID` and selects related rows from `Sales.OrderDetails`. If none exist, it prints a message and returns 1 (user-defined return code).

```
CREATE OR ALTER PROCEDURE Sales.GetOrderDetails
    @OrderID INT
AS
BEGIN
    SET NOCOUNT ON;
    IF NOT EXISTS (SELECT 1 FROM Sales.OrderDetails WHERE OrderID = @OrderID)
    BEGIN
        PRINT 'The OrderID ' + CAST(@OrderID AS VARCHAR) + ' does not exist.';
        RETURN 1;
    END
    SELECT OrderID, ProductID, UnitPrice, Quantity, Discount
      FROM Sales.OrderDetails
      WHERE OrderID = @OrderID;
END;
GO
```

1. **DeleteOrderDetails**– Takes `@OrderID, @ProductID`. It first validates that the referenced order and product exist (printing an error and returning -1 if invalid). If valid, it deletes the detail row.

```
CREATE OR ALTER PROCEDURE Sales.DeleteOrderDetails
    @OrderID INT,
    @ProductID INT
AS
BEGIN
    SET NOCOUNT ON;
    IF NOT EXISTS (SELECT 1 FROM Sales.Orders WHERE OrderID = @OrderID)
```

```
    BEGIN
        PRINT 'Invalid OrderID: ' + CAST(@OrderID AS VARCHAR);
        RETURN -1;
    END
    IF NOT EXISTS (SELECT 1 FROM Production.Product WHERE ProductID =
@ProductID)
    BEGIN
        PRINT 'Invalid ProductID: ' + CAST(@ProductID AS VARCHAR);
        RETURN -1;
    END
    IF NOT EXISTS (SELECT 1 FROM Sales.OrderDetails
                   WHERE OrderID = @OrderID AND ProductID = @ProductID)
    BEGIN
        PRINT 'OrderDetail (' + CAST(@OrderID AS VARCHAR) + ',' +
CAST(@ProductID AS VARCHAR) + ') does not exist.';
        RETURN -1;
    END

    DELETE FROM Sales.OrderDetails
    WHERE OrderID = @OrderID AND ProductID = @ProductID;
END;
GO
```

User-Defined Functions

Two simple scalar functions to format dates:

- **FormatDate_US(@d)**– returns `MM/DD/YYYY` by using `CONVERT(varchar, @d, 101)`. (Style **101** produces U.S. format mm/dd/yyyy [4].)
- **FormatDate_ISO(@d)**– returns `YYYYMMDD` by using `CONVERT(varchar, @d, 112)`. (Style **112** produces yyyymmdd [5].)

```
CREATE OR ALTER FUNCTION dbo.FormatDate_US(@d DATETIME)
RETURNS VARCHAR(10)
AS
BEGIN
    RETURN CONVERT(VARCHAR(10), @d, 101);  -- mm/dd/yyyy [4]
END;
GO

CREATE OR ALTER FUNCTION dbo.FormatDate_ISO(@d DATETIME)
RETURNS VARCHAR(8)
AS
BEGIN
    RETURN CONVERT(VARCHAR(8), @d, 112);   -- yyyymmdd [5]
```

```
END;
GO
```

Views

1. **vwCustomerOrders:** Joins customers, orders, order details, and products to show: `CompanyName, OrderID, OrderDate, ProductID, ProductName, Quantity, UnitPrice, TotalPrice` .(`TotalPrice` is calculated as `Quantity * UnitPrice` .)

2. **vwCustomerOrders_Yesterday:** Same columns as above, but filters to only those orders whose date is *yesterday*. (We compare `CAST(OrderDate AS DATE)` to `CAST(DATEADD(DAY, -1, GETDATE()) AS DATE)` .)

3. **MyProducts:** Joins `Production.Product` with `Supplier` and `Category` to show supplier and category names. It returns all **non-discontinued** products.

```
CREATE OR ALTER VIEW dbo.vwCustomerOrders AS
SELECT
c.CompanyName,
o.OrderID,
o.OrderDate,
d.ProductID,
    p.Name    AS ProductName,
d.Quantity,
d.UnitPrice,
(d.Quantity * d.UnitPrice) AS TotalPrice
FROM Sales.Orders AS o
JOIN Sales.Customers AS c ON o.CustomerID = c.CustomerID
JOIN Sales.OrderDetails AS d ON o.OrderID = d.OrderID
JOIN Production.Product AS p ON d.ProductID = p.ProductID; GO

CREATE OR ALTER VIEW dbo.vwCustomerOrders_Yesterday AS
SELECT *
FROM dbo.vwCustomerOrders
WHERE CAST(OrderDate AS DATE) = CAST(DATEADD(DAY, -1, GETDATE()) AS DATE); GO

CREATE OR ALTER VIEW dbo.MyProducts AS
SELECT
p.ProductID,
p.Name AS ProductName,
s.CompanyName AS SupplierName,
c.CategoryName,
p.ListPrice,
```

```
    p.UnitsInStock,
    p.ReorderLevel
FROM Production.Product AS p
LEFT JOIN Production.Supplier AS s ON p.SupplierID = s.SupplierID
LEFT JOIN Production.Category AS c ON p.CategoryID = c.CategoryID
WHERE p.Discontinued = 0;
GO
```

Triggers

1. **INSTEAD OF DELETE on `Sales.Orders`**: This trigger intercepts deletes on the `Orders` table. Instead of letting SQL Server cascade or block deletes, it explicitly deletes related `OrderDetails` first, then deletes the `Order` row. The trigger uses the `deleted` pseudo-table to find which `OrderID` was targeted [6].

2. **AFTER INSERT on `Sales.OrderDetails`**: This trigger fires after a new order detail is inserted. It checks the `inserted` pseudo-table against `Production.Product.UnitsInStock`. If any inserted detail has `Quantity > UnitsInStock`, it raises an error and rolls back the insert (using `RAISERROR` and `ROLLBACK` in the trigger). Otherwise, it deducts the inserted quantity from stock. (This enforces the same stock check as the stored procedure, as a safety net.)

```
-- INSTEAD OF DELETE on Orders
CREATE OR ALTER TRIGGER Sales.trg_Order_Delete
ON Sales.Orders
INSTEAD OF DELETE
AS
BEGIN
    -- Delete related details for each deleted order
    DELETE d
    FROM Sales.OrderDetails AS d
    JOIN deleted AS del ON d.OrderID = del.OrderID;
    -- Then delete the order itself
    DELETE o
    FROM Sales.Orders AS o
    JOIN deleted AS del ON o.OrderID = del.OrderID;
END;
GO

-- AFTER INSERT on OrderDetails
CREATE OR ALTER TRIGGER Sales.trg_OrderDetails_Insert
ON Sales.OrderDetails
AFTER INSERT
AS
BEGIN
    SET NOCOUNT ON;
```

```
    -- Abort if any inserted quantity exceeds stock
    IF EXISTS (
        SELECT 1
        FROM inserted i
        JOIN Production.Product p ON i.ProductID = p.ProductID
        WHERE i.Quantity > p.UnitsInStock
    )
    BEGIN
        RAISERROR('Insufficient stock for inserted order detail.', 16, 1);
        ROLLBACK TRANSACTION;
        RETURN;
    END
    -- Otherwise, update stock
    UPDATE p
    SET p.UnitsInStock = p.UnitsInStock - i.Quantity
    FROM Production.Product p
    JOIN inserted i ON p.ProductID = i.ProductID;
END;
GO
```

Key Points: Triggers can reference the **`inserted`** and **`deleted`** tables to see which rows were affected [6]. In an *INSTEAD OF DELETE* trigger, the `deleted` table contains the rows that were requested for deletion (hence we can delete related details first). The*AFTER INSERT* trigger uses `inserted` to validate stock immediately after the insert attempt.
