SQL2019 on RHEL8 Lab

Lab 1 : SQL2019 Install

##Pre-reqs:
sudo yum -y install python2
sudo yum -y install compat-openssl10
sudo alternatives --config python , select python 2

##Set up Repo & Download
sudo curl -o /etc/yum.repos.d/mssql-server.repo
https://packages.microsoft.com/config/rhel/8/mssql-server-2019.repo
sudo yum download mssql-server

##Install and Setup
sudo rpm -Uvh --nodeps mssql-server*rpm
sudo /opt/mssql/bin/mssql-conf setup
- Select 2 - Developer edition
- Accept (YES) License agreements
- DB password = r3dh4t1!

##Verify SQL is running:
systemctl status mssql-server

##Add remote connections to DB
sudo firewall-cmd --zone=public --add-port=1433/tcp --permanent
sudo firewall-cmd --reload

##Install SQL Command Line tools
sudo curl -o /etc/yum.repos.d/msprod.repo
https://packages.microsoft.com/config/rhel/8/prod.repo
sudo yum install -y mssql-tools unixODBC-devel
- Type YES for license

##Add mssql-tools to your PATH variable:
echo 'export PATH="$PATH:/opt/mssql-tools/bin"' >> ~/.bash_profile
echo 'export PATH="$PATH:/opt/mssql-tools/bin"' >> ~/.bashrc
source ~/.bashrc

```
##Connect to local DB
sqlcmd -S localhost -U SA -P r3dh4t1!

##Create Test DB & Table
  1> CREATE DATABASE TestDB
  2> SELECT Name from sys.Databases
  3> GO

  1> CREATE TABLE Inventory (id INT, name NVARCHAR(50), quantity INT)
  2> INSERT INTO Inventory VALUES (1, 'banana', 150); INSERT INTO Inventory VALUES (2,
'orange', 154);
  3> GO

## Select Data

  1> SELECT * FROM Inventory WHERE quantity > 152;
  2> GO

##Exit DB
  1> QUIT
```

Lab 2: Restore a SQL Server DB from Windows to RHEL

```
sudo yum install -y wget
cd /tmp
sudo wget
https://github.com/Microsoft/sql-server-samples/releases/download/adventureworks/Adventure
Works2014.bak
sudo mv AdventureWorks2014.bak /var/opt/mssql
sudo chown mssql:mssql /var/opt/mssql/AdventureWorks2014.bak
sudo chmod 755 /var/opt/mssql/AdventureWorks2014.bak

## Restore DB from Backup via Command Line
sqlcmd -S localhost -U SA -P r3dh4t1!

RESTORE FILELISTONLY FROM DISK = '/var/opt/mssql/AdventureWorks2014.bak
GO
```

##You will see 2 Files listed
AdventureWorks2014_Data
AdventureWorks2014_Log
So to Restore you will do:

```
RESTORE DATABASE AdventureWorks2014
FROM DISK = '/var/opt/mssql/AdventureWorks2014.bak'
WITH MOVE 'AdventureWorks2014_Data' TO
'/var/opt/mssql/data/AdventureWorks2014_Data.ndf',
MOVE 'AdventureWorks2014_Log' TO '/var/opt/mssql/data/AdventureWorks2014_Log.ldf'
GO
```

Verify you can see your restored DB

```
SELECT Name FROM sys.Databases
GO
```

Lab 3: Security best practices for SQL on RHEL

Create a login and a database user

##Grant others access to SQL Server by creating a login in the master database using the CREATE LOGIN statement. For example:

```
CREATE LOGIN student WITH PASSWORD = ‘r3dh4t1!’;
```

##To connect to a user-database, a login needs a corresponding identity at the database level, called a database user. Users are specific to each database and must be separately created in each database to grant them access.

```
USE AdventureWorks2014;
GO
CREATE USER student;
CREATE USER Jerry;
GO
```

##You can authorize other logins to create a more logins by granting them the ALTER ANY LOGIN permission. Inside a database, you can authorize other users to create more users by granting them the ALTER ANY USER permission. For example:

```
GRANT ALTER ANY LOGIN TO student;
GO

USE AdventureWorks2014;
GO
GRANT ALTER ANY USER TO Jerry;
GO
```

Now the login Larry can create more logins, and the user Jerry can create more users.

Granting access with least privileges

The first people to connect to a user-database will be the administrator and database owner accounts. However these users have all the permissions available on the database. This is more permission than most users should have.
When you are just getting started, you can assign some general categories of permissions by using the built-in *fixed database roles*. For example, the db_datareader fixed database role can read all tables in the database, but make no changes. Grant membership in a fixed database role by using the ALTER ROLE statement. The following example add the user Jerry to the db_datareader fixed database role.

```
USE AdventureWorks2014;
GO

ALTER ROLE db_datareader ADD MEMBER Jerry;
```

Later, when you are ready to configure more precise access to your data (highly recommended), create your own user-defined database roles using CREATE ROLE statement. Then assign specific granular permissions to you custom roles.
For example, the following statements create a database role named Sales, grants the Sales group the ability to see, update, and delete rows from the Orders table, and then adds the user Jerry to the Sales role.

```
CREATE ROLE Sales;
GRANT SELECT ON Object::Sales TO Orders;
GRANT UPDATE ON Object::Sales TO Orders;
GRANT DELETE ON Object::Sales TO Orders;
ALTER ROLE Sales ADD MEMBER Jerry;
```

Configure row-level security

#[Row-Level Security](../relational-databases/security/row-level-security.md) enables you to restrict access to rows in a database based on the user executing a query. This feature is useful for scenarios like ensuring that customers can only access their own data or that workers can only access data that is pertinent to their department.

#The following steps walk through setting up two Users with different row-level access to the `Sales.SalesOrderHeader` table.

#Create two user accounts to test the row level security:

```
USE AdventureWorks2014;
GO

CREATE USER Manager WITHOUT LOGIN;

CREATE USER SalesPerson280 WITHOUT LOGIN;
```

#Grant read access on the `Sales.SalesOrderHeader` table to both users:

```
GRANT SELECT ON Sales.SalesOrderHeader TO Manager;
GRANT SELECT ON Sales.SalesOrderHeader TO SalesPerson280;
```

#Create a new schema and inline table-valued function. The function returns 1 when a row in the `SalesPersonID` column matches the ID of a `SalesPerson` login or if the user executing the query is the Manager user.

```
CREATE SCHEMA Security;
GO

CREATE FUNCTION Security.fn_securitypredicate(@SalesPersonID AS int)
    RETURNS TABLE
WITH SCHEMABINDING
AS
   RETURN SELECT 1 AS fn_securitypredicate_result
WHERE ('SalesPerson' + CAST(@SalesPersonId as VARCHAR(16)) = USER_NAME())
    OR (USER_NAME() = 'Manager');
```

#Create a security policy adding the function as both a filter and a block predicate on the table:

```
CREATE SECURITY POLICY SalesFilter
ADD FILTER PREDICATE Security.fn_securitypredicate(SalesPersonID)
ON Sales.SalesOrderHeader,
ADD BLOCK PREDICATE Security.fn_securitypredicate(SalesPersonID)
ON Sales.SalesOrderHeader
WITH (STATE = ON);
```

#Execute the following to query the `SalesOrderHeader` table as each user. Verify that `SalesPerson280` only sees the 95 rows from their own sales and that the `Manager` can see all the rows in the table.

```
EXECUTE AS USER = 'SalesPerson280';
SELECT * FROM Sales.SalesOrderHeader;
REVERT;
EXECUTE AS USER = 'Manager';
SELECT * FROM Sales.SalesOrderHeader;
REVERT;
```

#Alter the security policy to disable the policy. Now both users can access all rows.

```
ALTER SECURITY POLICY SalesFilter
WITH (STATE = OFF);
```

Enable dynamic data masking

#[Dynamic Data Masking](../relational-databases/security/dynamic-data-masking.md) enables you to limit the exposure of sensitive data to users of an application by fully or partially masking certain columns.

#Use an `ALTER TABLE` statement to add a masking function to the `EmailAddress` column in the `Person.EmailAddress` table:

```
USE AdventureWorks2014;
GO ALTER TABLE Person.EmailAddress
ALTER COLUMN EmailAddress
ADD MASKED WITH (FUNCTION = 'email()');
```

#Create a new user `TestUser` with `SELECT` permission on the table, then execute a query as `TestUser` to view the masked data:

```
CREATE USER TestUser WITHOUT LOGIN;
```

```
GRANT SELECT ON Person.EmailAddress TO TestUser;
EXECUTE AS USER = 'TestUser';
SELECT EmailAddressID, EmailAddress FROM Person.EmailAddress;
REVERT;
```

Verify that the masking function changes the email address in the first record from:

EmailAddressID	EmailAddress
1	ken0@adventure-works.com

into

EmailAddressID	EmailAddress
1	kXXX@XXXX.com

Enable Transparent Data Encryption

#One threat to your database is the risk that someone will steal the database files off of your hard-drive. This could happen with an intrusion that gets elevated access to your system, through the actions of a problem employee, or by theft of the computer containing the files (such as a laptop).

#Transparent Data Encryption (TDE) encrypts the data files as they are stored on the hard drive. The master database of the SQL Server database engine has the encryption key, so that the database engine can manipulate the data. The database files cannot be read without access to the key. #High-level administrators can manage, backup, and recreate the key, so the database can be moved, but only by selected people. When TDE is configured, the `tempdb` database is also automatically encrypted.

#Since the Database Engine can read the data, Transparent Data Encryption does not protect against unauthorized access by administrators of the computer who can directly read memory, or access SQL Server through an administrator account.

Configure TDE

#- Create a master key
#- Create or obtain a certificate protected by the master key
#- Create a database encryption key and protect it by the certificate
#- Set the database to use encryption

#Configuring TDE requires `CONTROL` permission on the master database and `CONTROL` permission on the user database. Typically an administrator configures TDE.

#The following example illustrates encrypting and decrypting the `AdventureWorks2014` database using a certificate installed on the server named `MyServerCert`.

```
USE master;
GO
CREATE MASTER KEY ENCRYPTION BY PASSWORD = '**********';
GO
CREATE CERTIFICATE MyServerCert WITH SUBJECT = 'My Database Encryption Key Certificate';
GO
USE AdventureWorks2014;   GO
CREATE DATABASE ENCRYPTION KEY
WITH ALGORITHM = AES_256
ENCRYPTION BY SERVER CERTIFICATE MyServerCert;
GO
ALTER DATABASE AdventureWorks2014
SET ENCRYPTION ON;
```

#To remove TDE, execute `ALTER DATABASE AdventureWorks2014 SET ENCRYPTION OFF;`

#The encryption and decryption operations are scheduled on background threads by SQL Server. You can view the status of these operations using the catalog views and dynamic management views in the list that appears later in this topic.

[!WARNING]
Backup files of databases that have TDE enabled are also encrypted by using the database encryption key. As a result, when you restore these backups, the certificate protecting the database encryption key must be available. This means that in addition to backing up the database, you have to make sure that you maintain backups of the server certificates to prevent data loss. Data loss will result if the certificate is no longer available. For more information, see [SQL Server Certificates and Asymmetric Keys](../relational-databases/security/sql-server-certificates-and-asymmetric-keys.md).

Configure backup encryption
#SQL Server has the ability to encrypt the data while creating a backup. By specifying the encryption algorithm and the encryptor (a certificate or asymmetric key) when creating a backup, you can create an encrypted backup file.

[!WARNING]
It is very important to back up the certificate or asymmetric key, and preferably to a different location than the backup file it was used to encrypt. Without the certificate or asymmetric key, you cannot restore the backup, rendering the backup file unusable.

The following example creates a certificate, and then creates a backup protected by the certificate.

```
USE master;   GO   CREATE CERTIFICATE BackupEncryptCert   WITH SUBJECT = 'Database backups';   GO BACKUP DATABASE [AdventureWorks2014]   TO DISK = N'/var/opt/mssql/backups/AdventureWorks2014.bak'
WITH
COMPRESSION,
ENCRYPTION
(
ALGORITHM = AES_256,
SERVER CERTIFICATE = BackupEncryptCert
),
STATS = 10
GO
```

SQL on RHEL Lab Completed!