Microsoft

Data Encryption and Security

Module 3

CONDITIONS AND TERMS OF USE:

Learning Units covered in this Module

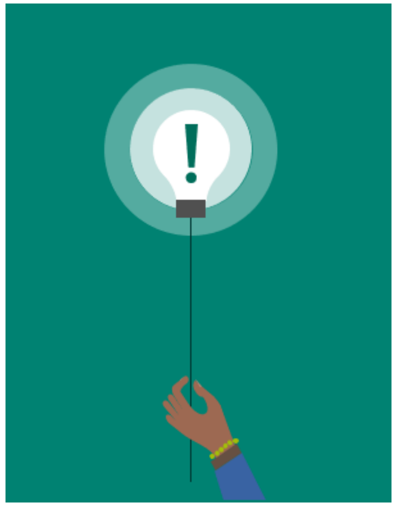
- Lesson 1: Azure Key Vault
- Lesson 2: Transparent Data Encryption (TDE)
- Lesson 3: Always Encrypted
- Lesson 4: Row-Level Security
- Lesson 5: Dynamic Data Masking

Lesson 1: Azure Key Vault

Objectives

After completing this learning, you will be able to:

- Understand what is Azure Key Vault
- Understand best practices



What is Azure Key Vault?

- Azure service for securely storing and accessing secrets
- Store Secrets, Keys, and Certificates
- Access to Vault require Authentication and Authorization
- Monitor Activity

Azure Key Vault

Secrets Management

- Store and tightly control access to tokens, passwords, certificates, API Keys, and Secrets

Key Management

- Accessible to create and control encryption keys to encrypt data in minutes
- Applications do not have direct access to Keys

Certificate Management

- Simplify and automate tasks for SSL/TLS certificates used by Azure and Internal connected resources

Azure Key Vault Best Practices

Control Access	Lock down subscription, RG, and Key Vaults (RBAC) Access Policy for all Vaults Firewall and VNET Service End Points
Separate Key Vault	Use separate Key Vault for Prod, QA, and DEV
Backup	Take backups before Update/Delete/Create Objects
Logging	Turn on Logging Setup Alerts
Recovery	Enable Soft Delete Enable Purge Protection

Demonstration

Azure Key Vault

- Create Azure Key Vault
- Update Permissions
- Create secret
- Backup

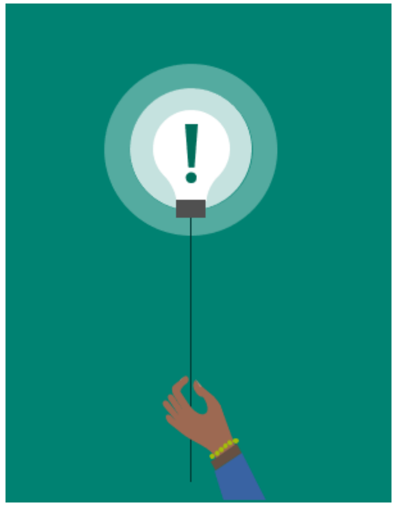
Questions?

Lesson 2: Transparent Data Encryption (TDE)

Objectives

After completing this learning, you will be able to:

- Understand Transparent Data Encryption (TDE)
- Types on Encryption Keys
- Enable TDE in SQL Managed Instance



Understanding TDE Functionality

Data is encrypted at rest.

Encryption keys are managed by Azure.

Performs real-time I/O encryption and decryption of the data at the page level.

Each page is decrypted when it's read into memory and then encrypted before being written to disk.

TDE is enabled for all newly deployed Azure SQL MIs.

No need for application change.

Bring You Own Key (BYOK) supported with Azure Key Vault

Encryption Keys

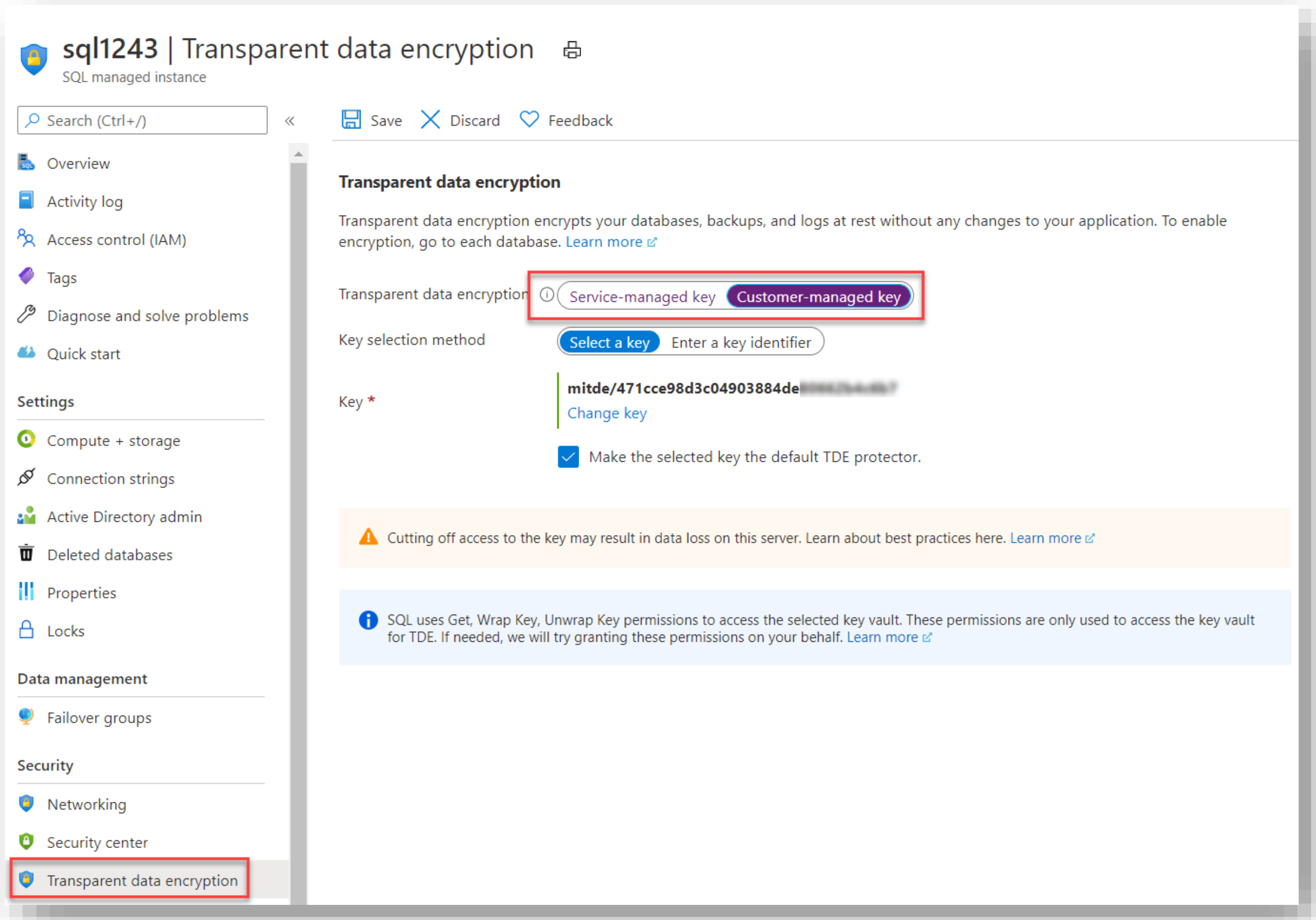
Service-managed transparent data encryption

- The database encryption key is protected by a built-in server certificate.
- Unique for each server.
- Primary and geo-secondary database are protected by the primary database's parent server key.
- Microsoft automatically rotates these certificates at least every 90 days.

Bring Your Own Key

- Take control over your transparent data encryption keys and control who can access them and when.
- Azure Key Vault.
- You set the asymmetric key at the server level, and all databases under that server inherit it.
- You can control key management tasks such as key rotations and key vault permissions.

Enable TDE Using Azure Portal



sql1243 | Transparent data encryption

SQL managed instance

Search (Ctrl+/)

Save | Discard | Feedback

- Overview
- Activity log
- Access control (IAM)
- Tags
- Diagnose and solve problems
- Quick start

Settings

- Compute + storage
- Connection strings
- Active Directory admin
- Deleted databases
- Properties
- Locks

Data management

- Failover groups

Security

- Networking
- Security center
- Transparent data encryption

Transparent data encryption

Transparent data encryption encrypts your databases, backups, and logs at rest without any changes to your application. To enable encryption, go to each database. Learn more

Transparent data encryption: Service-managed key [✓ Customer-managed key]

Key selection method: [✓ Select a key] Enter a key identifier

Key *: **mitde/471cce98d3c04903884de**[redacted]
Change key

✓ Make the selected key the default TDE protector.

⚠ Cutting off access to the key may result in data loss on this server. Learn about best practices here. Learn more

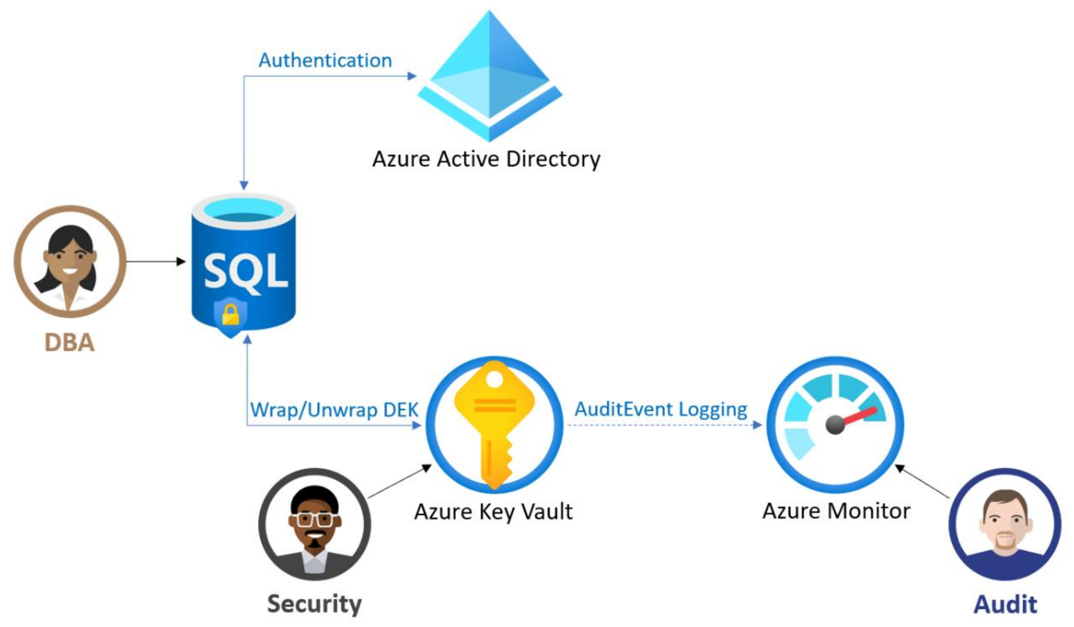
ⓘ SQL uses Get, Wrap Key, Unwrap Key permissions to access the selected key vault. These permissions are only used to access the key vault for TDE. If needed, we will try granting these permissions on your behalf. Learn more

TDE with customer-managed key (BYOK)

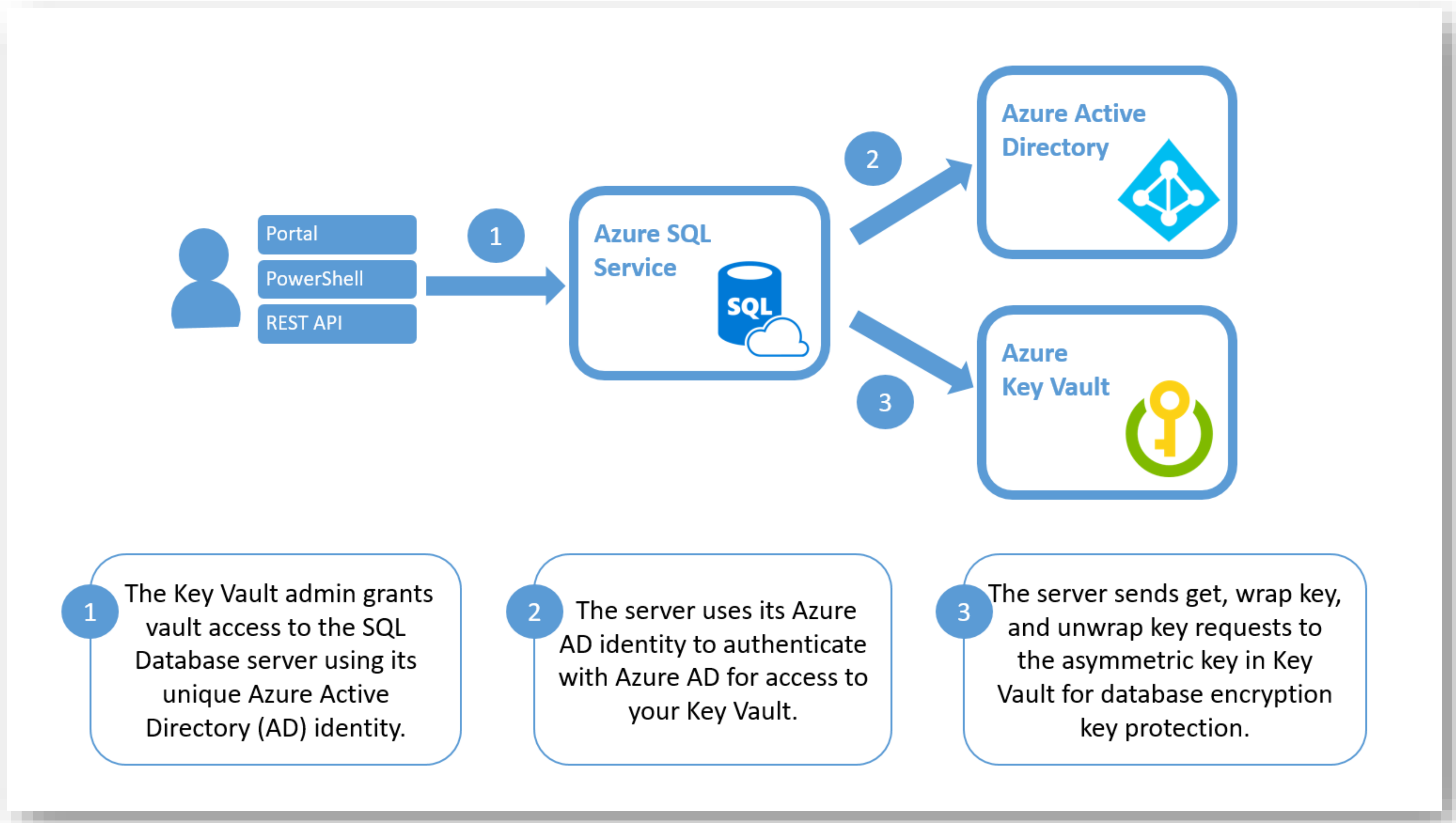
You are responsible for and in a full control of a key lifecycle management (key creation, upload, rotation, deletion), key usage permissions, and auditing of operations on keys.

The key used for encryption of the Database Encryption Key (DEK), called TDE protector, is a customer-managed asymmetric key stored in a customer-owned and customer-managed Azure Key Vault (AKV), a cloud-based external key management system.

TDE protector is set at the instance level and is inherited by all encrypted databases associated with that server.



TDE with Customer-Managed Keys in Azure Key Vault

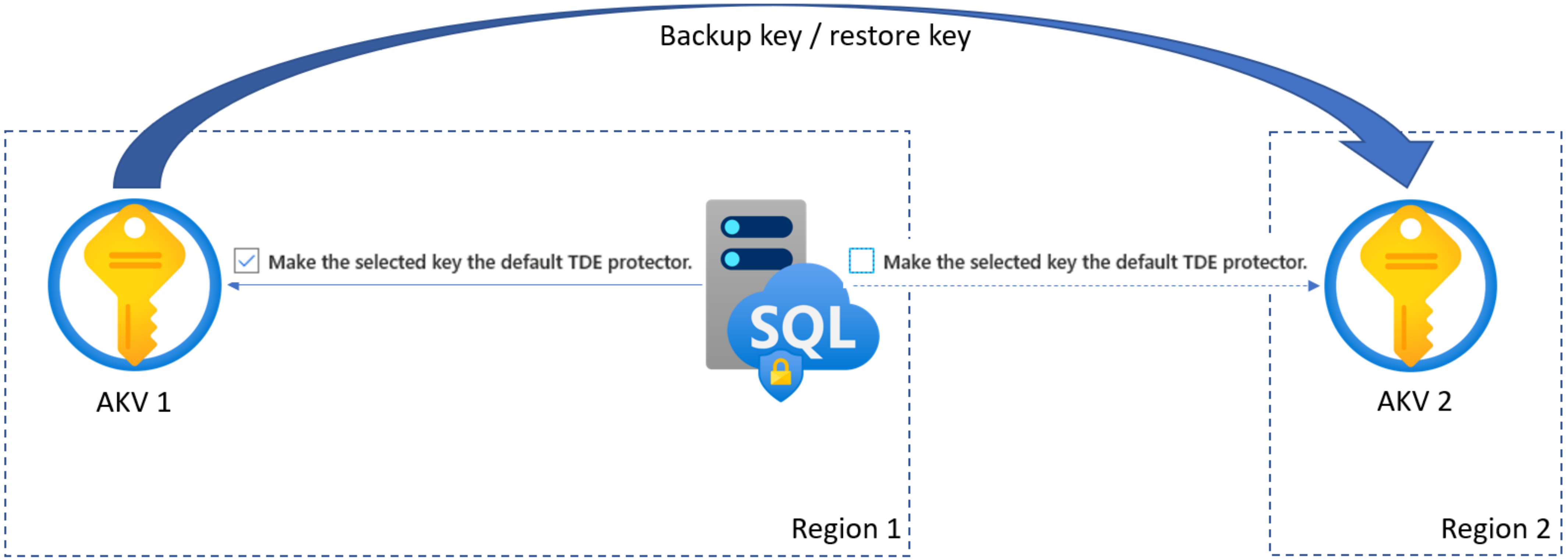


1. The Key Vault admin grants vault access to the SQL Database server using its unique Azure Active Directory (AD) identity.

2. The server uses its Azure AD identity to authenticate with Azure AD for access to your Key Vault.

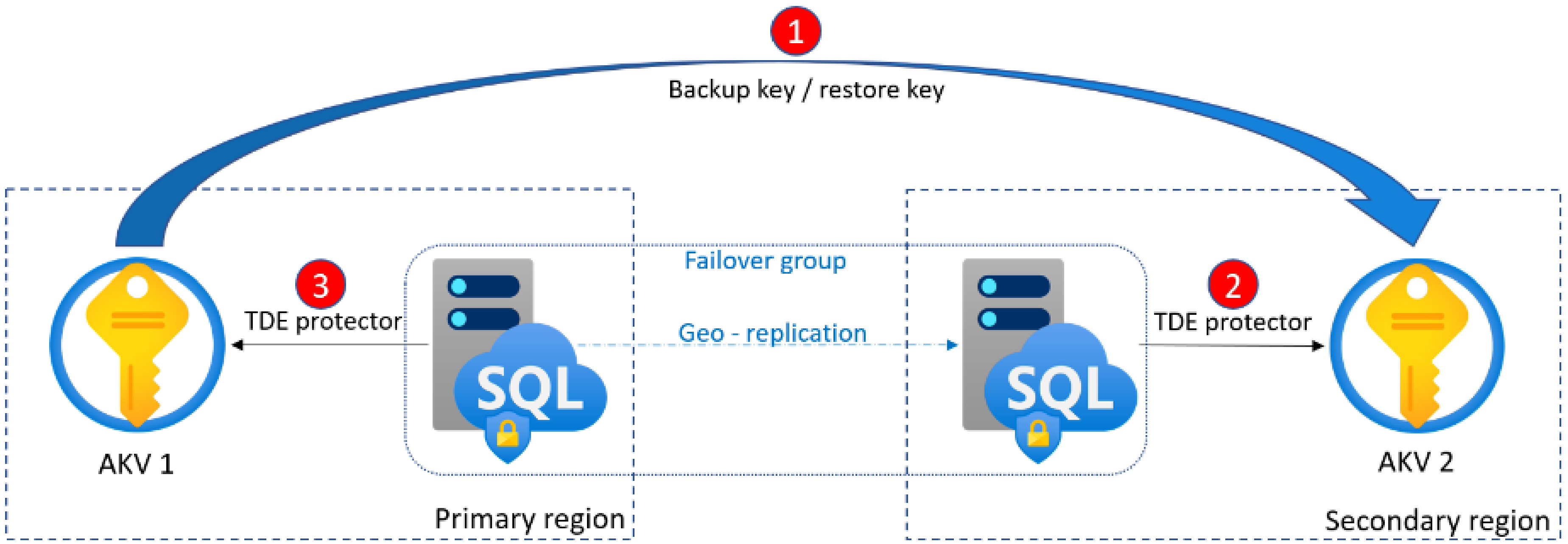
3. The server sends get, wrap key, and unwrap key requests to the asymmetric key in Key Vault for database encryption key protection.

High Availability with customer-managed TDE


Backup key / restore key
☑ Make the selected key the default TDE protector.
☐ Make the selected key the default TDE protector.
SQL
AKV 1
AKV 2
Region 1
Region 2

Geo-DR and customer-managed TDE

- All key vaults involved must have same properties, and same access rights for respective servers.
- All key vaults involved must contain identical key material. It applies not just to the current TDE protector, but to the all previous TDE protectors that may be used in the backup files.
- Both initial setup and rotation of the TDE protector must be done on the secondary first, and then on primary.



Demonstration

Implement TDE using Azure Portal and T-SQL Code

- Enable TDE With Bring Your Own Key using Azure Portal.

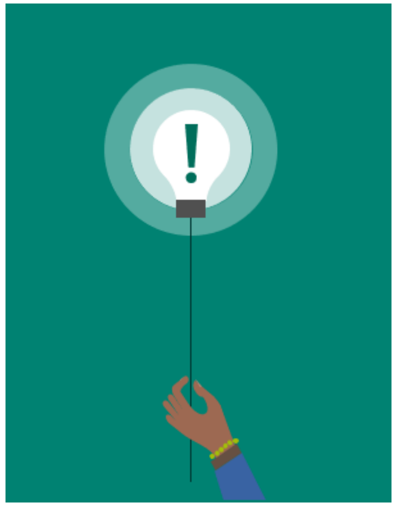
Questions?

Lesson 3: Always Encrypted

Objectives

After completing this learning, you will be able to:

- Understand Always Encrypted
- Encryption Methodologies

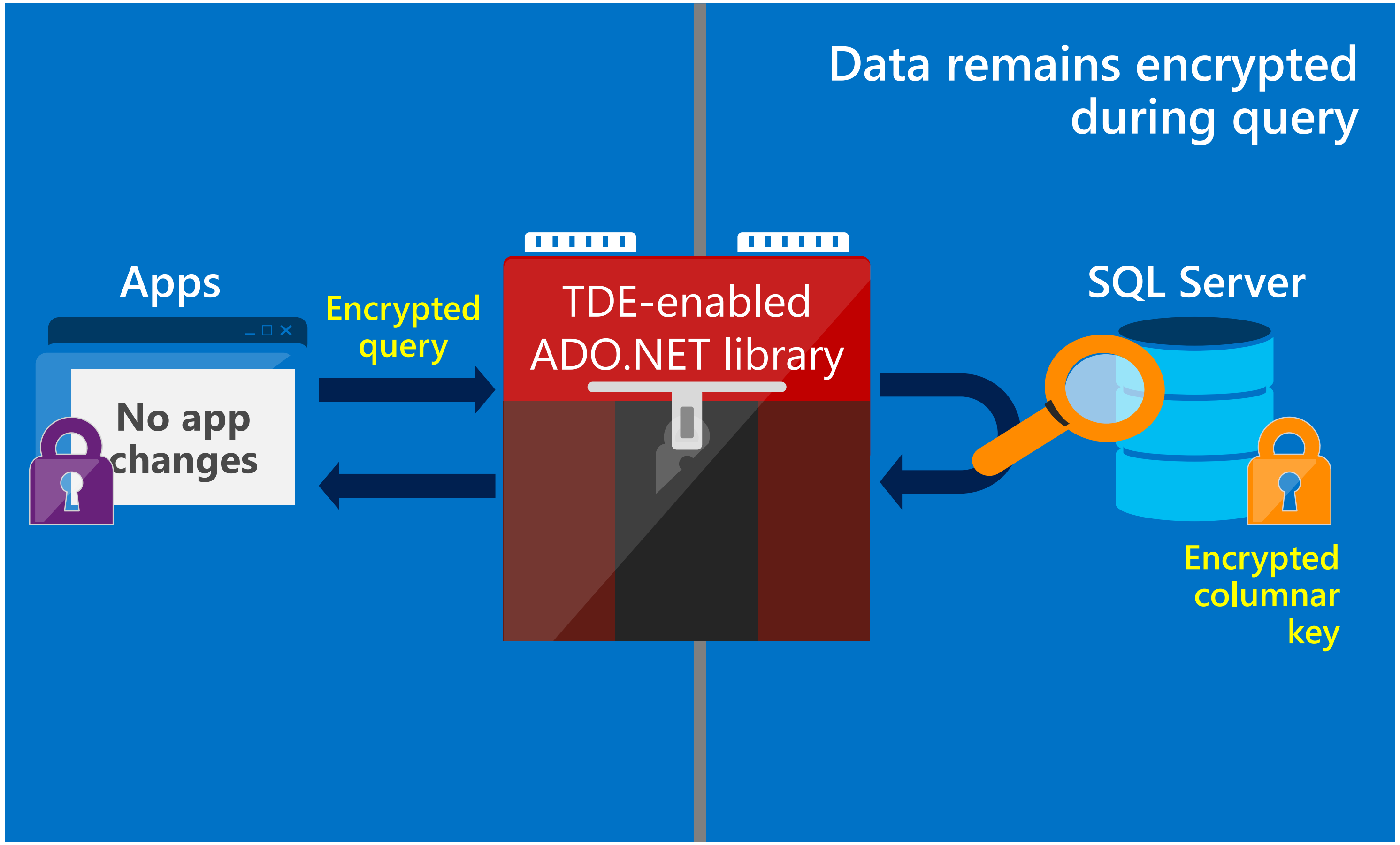


Always Encrypted

Always Encrypted allows clients to encrypt sensitive data inside client applications and never reveal the encryption keys to SQL Database.

As a result, Always Encrypted provides a separation between those who own the data (and can view it) and those who manage the data (but should have no access).

Understanding Always Encrypted Functionality



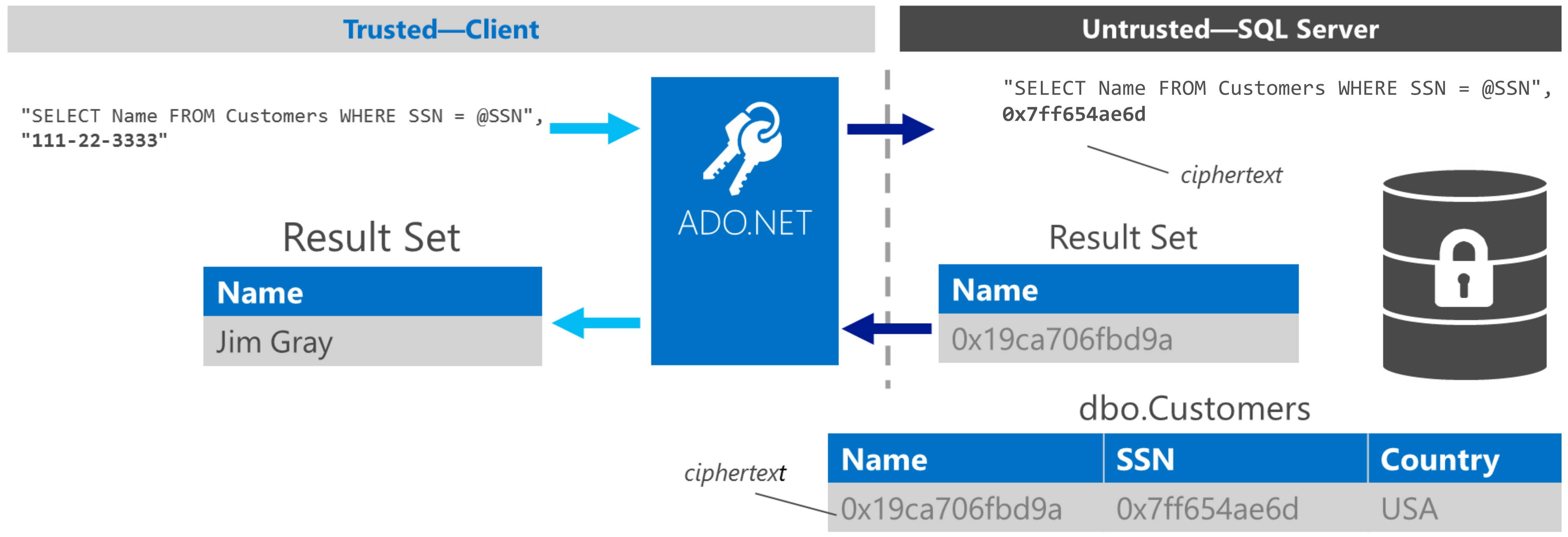
Capability

- Transparent client-side encryption, while SQL Server executes T-SQL queries on encrypted data.

Benefits

- Sensitive data remains encrypted and query-able at all times.
- Unauthorized users never have access to data or keys.
- No changes to applications are necessary.

Understanding Always Encrypted Functionality (Contd.)



```
"SELECT Name FROM Customers WHERE SSN = @SSN",
"111-22-3333"
```

```
"SELECT Name FROM Customers WHERE SSN = @SSN",
0x7ff654ae6d
```

Result Set

Name
Jim Gray

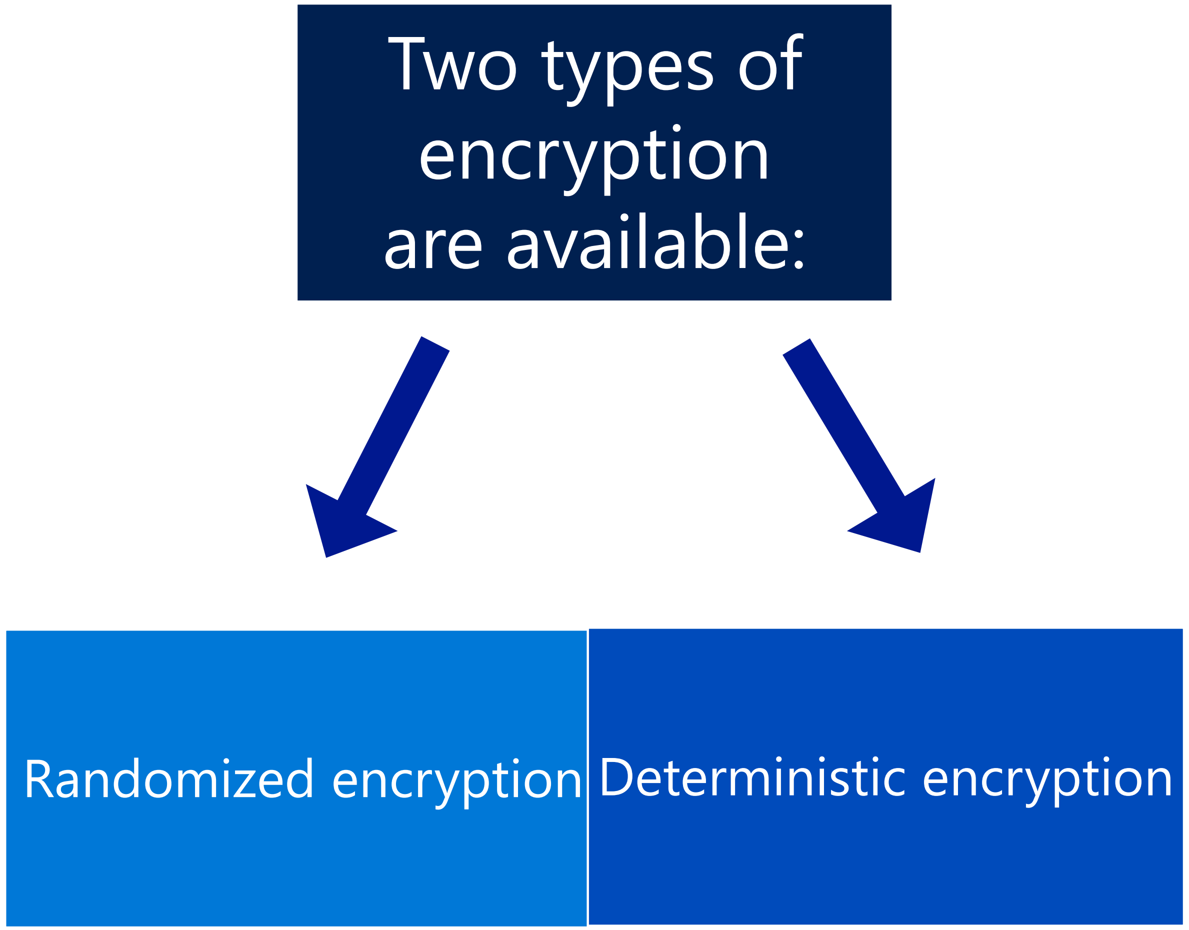
Result Set

Name
0x19ca706fbd9a

dbo.Customers

Name	SSN	Country
0x19ca706fbd9a	0x7ff654ae6d	USA

Encryption Methodologies



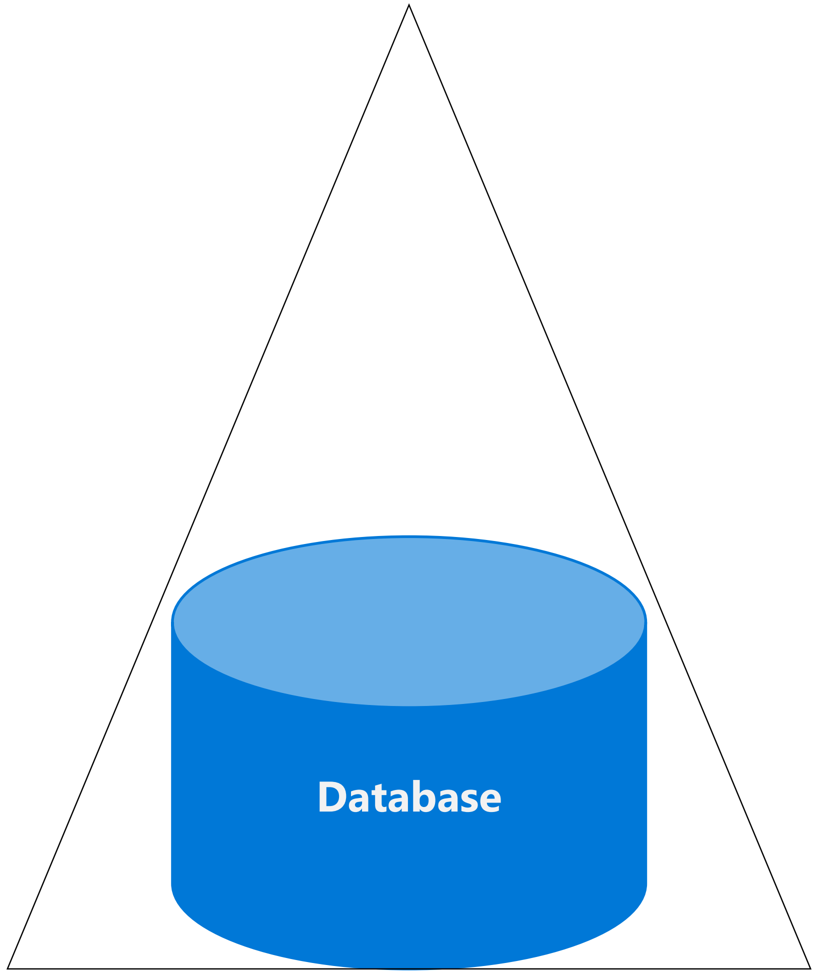
Randomized encryption

- Encrypt('123-45-6789') = 0x17cfd50a
- Repeat: Encrypt('123-45-6789') = 0x9b1fcf32
- Allows for transparent retrieval of encrypted data **but no operations.**
- More secure

Deterministic encryption

- Encrypt('123-45-6789') = 0x85a55d3f
- Repeat: Encrypt('123-45-6789') = 0x85a55d3f
- Allows for transparent retrieval of encrypted data **and quality.**
- **Comparison** (for example, in WHERE clauses and joins, distinct, group by).

Enabling Always Encrypted on Azure SQL MI



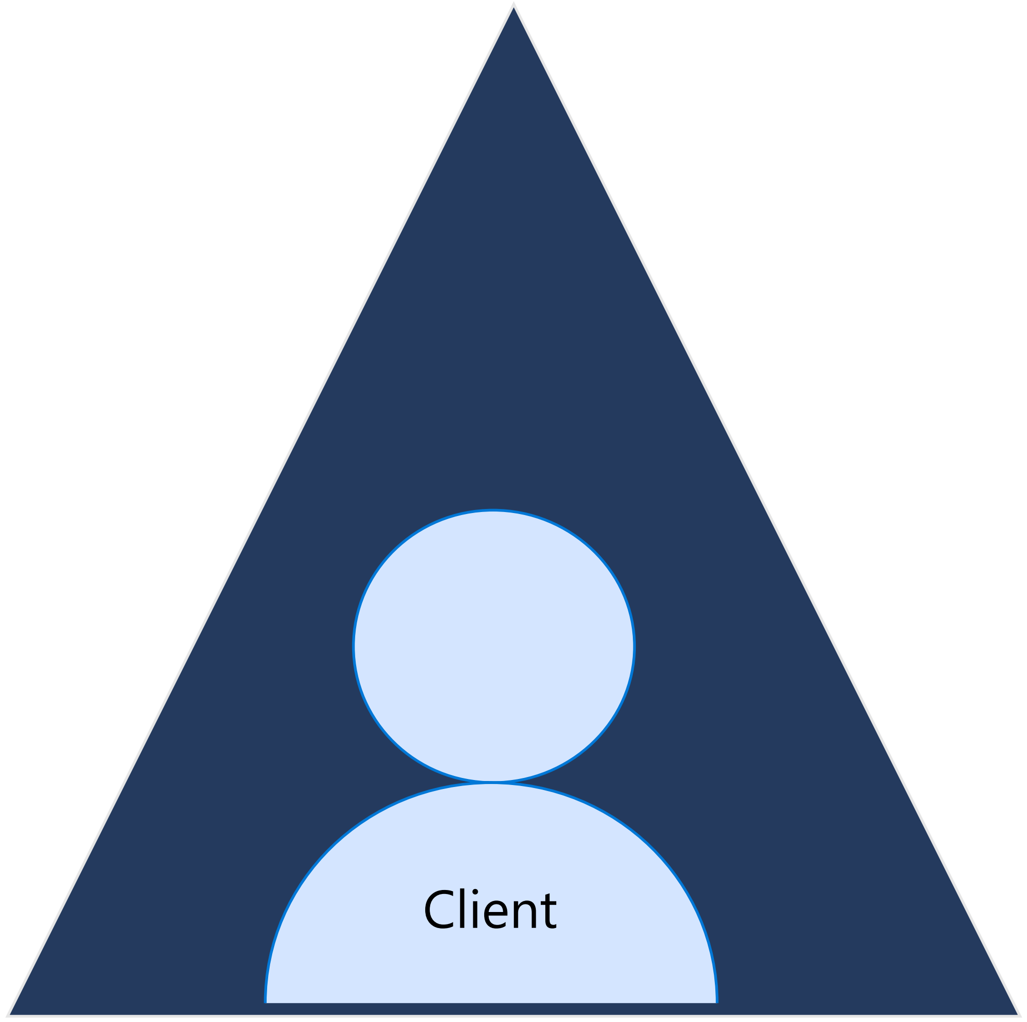
Create a local, self-signed certificate which will act as a Column Master Key (CMK).

Create a CMK store definition object in the database, which will store the information about the location of the CMK.

Create a Column Encryption Key (CEK) to encrypt columns in tables

Encrypt columns in the table

Enabling Always Encrypted on Azure SQL MI (contd.)



Same local stored certificate.

“Column Encryption Setting=Enabled;” in connection string.

ADO.NET driver

Enabling Always Encrypted using Azure Key Vault

1. Create Azure Key Vault, validate permissions, and configure firewall.
2. Create a key which will act as a Column Master Key (CMK).
3. Create a CMK store definition object in the database using Key in Azure Key Vault.
4. Create a Column Encryption Key (CEK) to encrypt columns in tables.
5. Encrypt columns in the table.

Enabling Always Encrypted using Azure Key Vault (contd.)

Modify Connection string to utilize Key from Azure Key Vault

Register the Azure Key Vault Provider

“Column Encryption Setting=Enabled;”
 in connection string.

ADO.Net Driver

* Azure Key Vault does not support local certificates. If application code cannot be modified, Azure Key Vault cannot be used

Demonstration

Enable Always Encrypted

- Enable Always Encrypted
- Select data through Application.

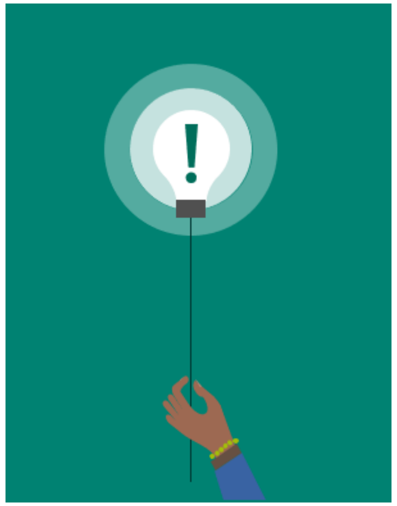
Questions?

Lesson 4: Row-Level Security

Objectives

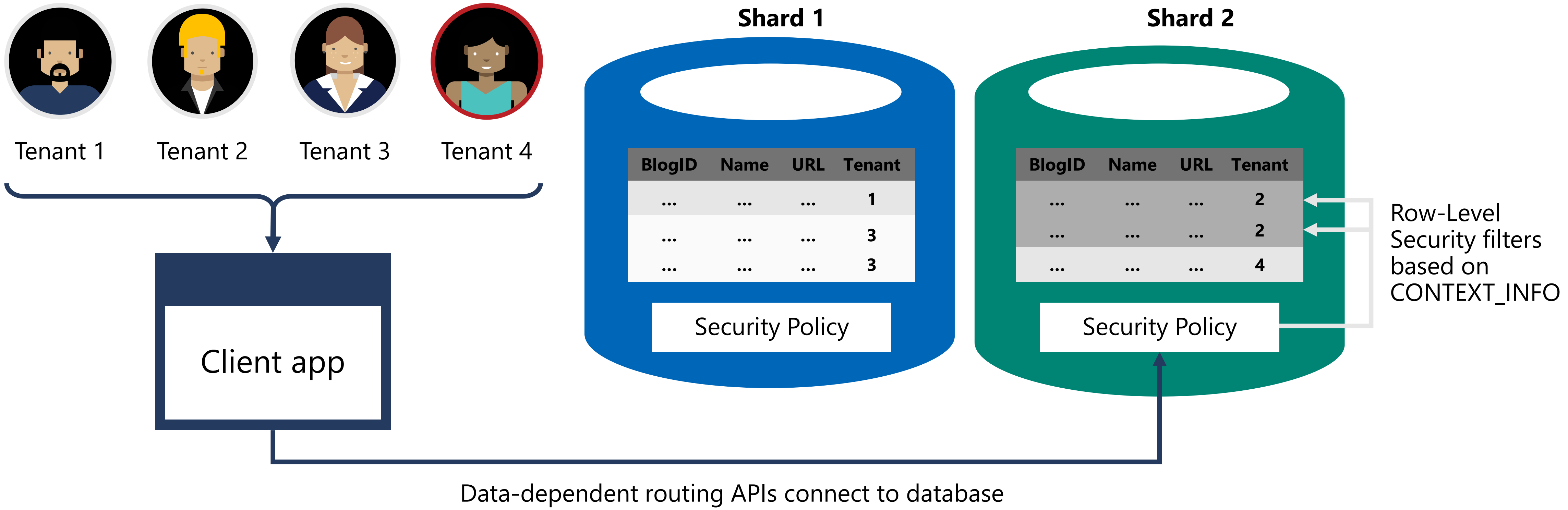
After completing this learning, you will be able to:

- Understand the Row Level Security feature in SQL Managed Instance
- Use case scenarios
- Benefits
- Implement Row Level Security



Row Level Security Introduction

RLS restricts which users can view which data in a table, based on a function


Tenant 1
Tenant 2
Tenant 3
Tenant 4
Shard 1
Shard 2
Client app
Security Policy
Security Policy
Row-Level Security filters based on CONTEXT_INFO
Data-dependent routing APIs connect to database

Shard 1

BlogID	Name	URL	Tenant
...	...	...	1
...	...	...	3
...	...	...	3

Shard 2

BlogID	Name	URL	Tenant
...	...	...	2
...	...	...	2
...	...	...	4

Row Level Security Scenarios

A hospital can restrict doctors and nurses to only view data about their specific patients.

A bank can restrict access to data based on the location of their branch offices.

A bicycle company can restrict sales leads to only specific salespeople.

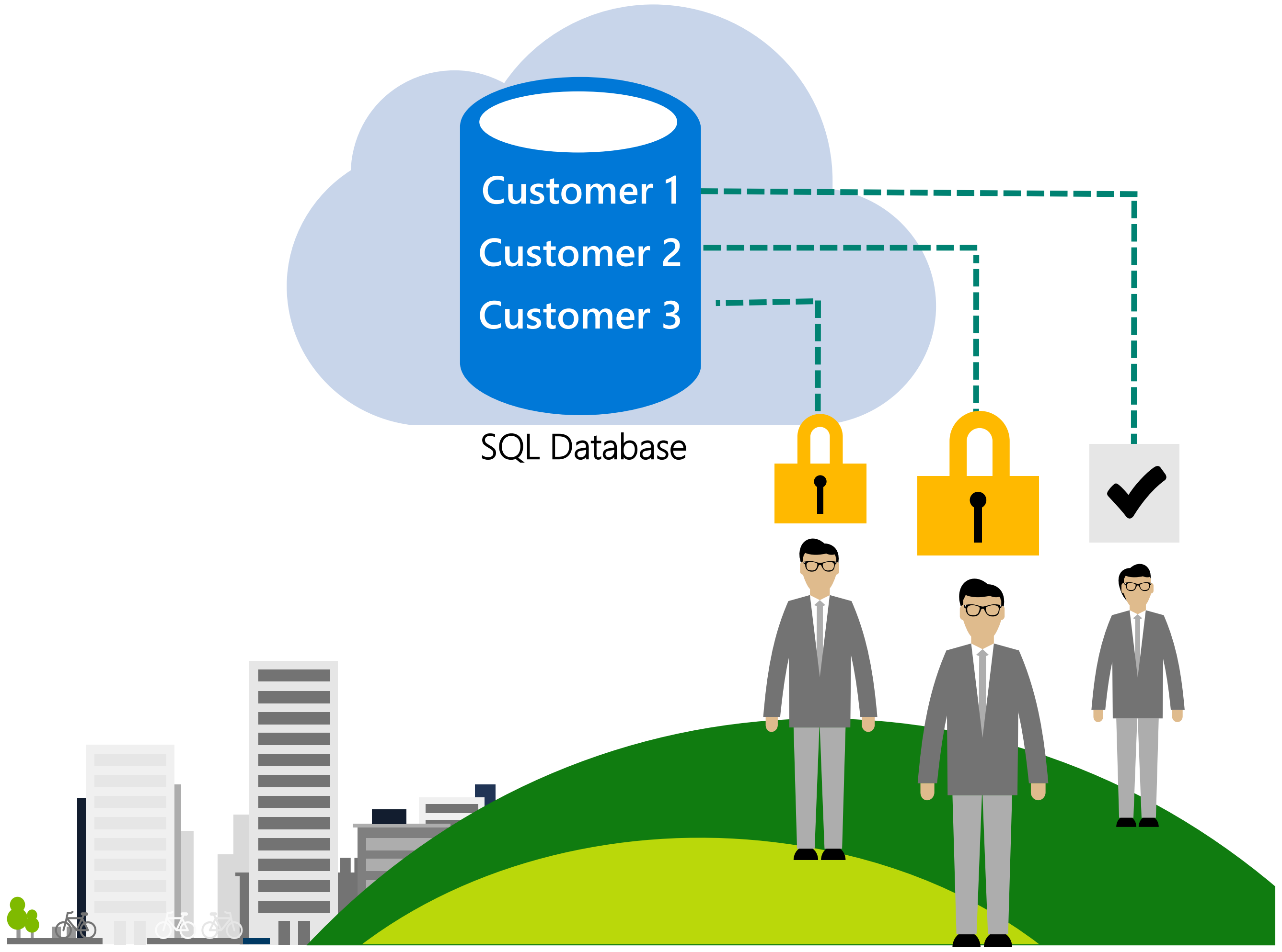
Need for Row-Level Security (RLS)

Enable fine-grained access control over specific rows in database table

Help prevent unauthorized access when multiple users share same tables.

Administer through SSMS or SSDT

Use enforcement logic inside database and schema-bound to the table



Row Level Security Benefits

Fine-grained access control

- Helps keep multi-tenant databases secure by limiting access by other users who share same tables

Application transparency

- Works transparently at query time, no app changes needed
- Offers compatibility with RLS in other leading products

Centralized security logic

- Increases security with enforcement logic residing inside database
- Reduces application maintenance and complexity

Row Level Security Concepts

Predicate function

- User-defined, inline table-valued function (iTVF) implementing security logic
- Can be arbitrarily complicated containing joins with other tables

Security predicate

- Predicate function bound to particular table, applying it for all queries
- Two types: filter predicates and blocking predicates

Security policy

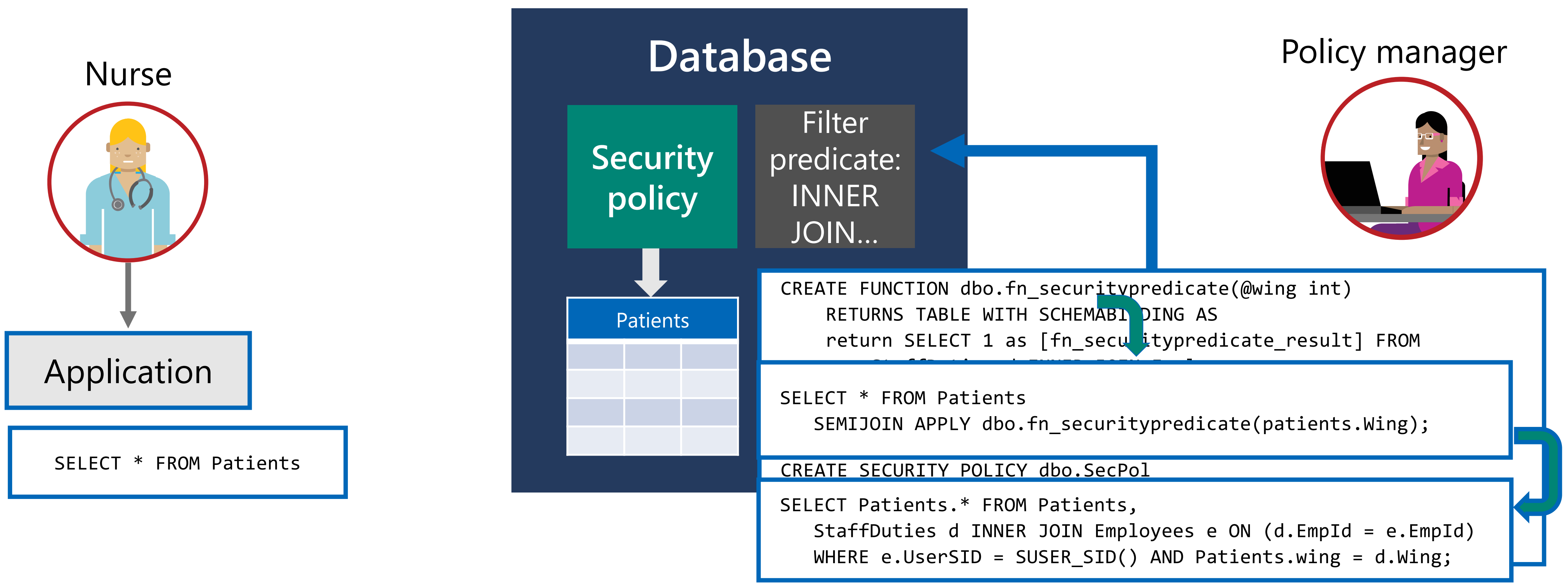
- Collection of security predicates for managing security across multiple tables

Performance?

Inline functions get optimized to provide comparable performance to views

```
CREATE SECURITY POLICY mySecurityPolicy
    ADD FILTER PREDICATE dbo.fn_securitypredicate(wing, startTime, endTime)
    ON dbo.patients
```

Row Level Security in Three Steps



```
SELECT * FROM Patients
```

```
CREATE FUNCTION dbo.fn_securitypredicate(@wing int)
    RETURNS TABLE WITH SCHEMABINDING AS
    return SELECT 1 as [fn_securitypredicate_result] FROM
```

```
SELECT * FROM Patients
   SEMIJOIN APPLY dbo.fn_securitypredicate(patients.Wing);
```

```
CREATE SECURITY POLICY dbo.SecPol
```

```
SELECT Patients.* FROM Patients,
   StaffDuties d INNER JOIN Employees e ON (d.EmpId = e.EmpId)
   WHERE e.UserSID = SUSER_SID() AND Patients.wing = d.Wing;
```

One

Three

Policy manager creates filter predicate and security policy in T-SQL, binding predicate to patient's table

[illegible] filter predicate

Create security policy for RLS

```
 -- The following syntax creates a security policy with a filter predicate for the Customer table, and
leaves the security policy disabled
 CREATE SECURITY POLICY [FederatedSecurityPolicy]
         ADD FILTER PREDICATE [rls].[fn_securitypredicate]([CustomerId])
         ON [dbo].[Customer];



 -- Create a new schema and predicate function, which will use the application user ID stored in
CONTEXT_INFO to filter rows.
 CREATE FUNCTION rls.fn_securitypredicate (@AppUserId int)
         RETURNS TABLE
         WITH SCHEMABINDING
 AS
 RETURN (
 SELECT 1 AS fn_securitypredicate_result
 WHERE
         DATABASE_PRINCIPAL_ID() = DATABASE_PRINCIPAL_ID('dbo') -- application context
         AND CONTEXT_INFO() = CONVERT(VARBINARY(128), @AppUserId);
 GO
```

Security predicates

- Access to row-level data restricted by security predicate
- Defined as inline table-valued function, invoked and enforced by security policy
- Two types of security predicates
  - Filter: silently filter rows available to read operations
  - Block: block write operations that violate predicate

Row Level Security Common Use Cases

- Traditional RLS workloads
- Multi-tenant databases
- Reporting, analytics, data warehousing

Demonstration

Row Level Security

- Creating a demo database and required objects
- Implementing and Testing Row Level Security

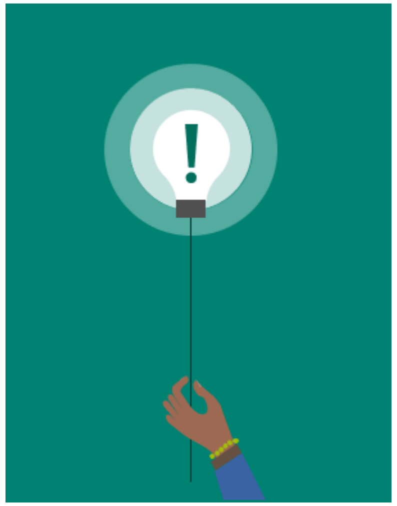
Questions?

Lesson 5: Dynamic Data Masking

Objectives

After completing this learning, you will be able to:

- Understand Dynamic Data Masking
- Use case scenarios
- Dynamic Data Masking functions
- Implement Data Dynamic Data Masking



Dynamic Data Masking

- Dynamic Data Masking is a policy-based security feature that helps to limit the exposure of data in a database by returning masked data to non-privileged users who run queries over designated database fields.

Dynamic Data Masking

Prevent abuse of sensitive data by hiding it from users

Policy-driven at table and column level for defined set of users

Applied in real time to query results based on policy

Multiple masking functions available for various sensitive data categories

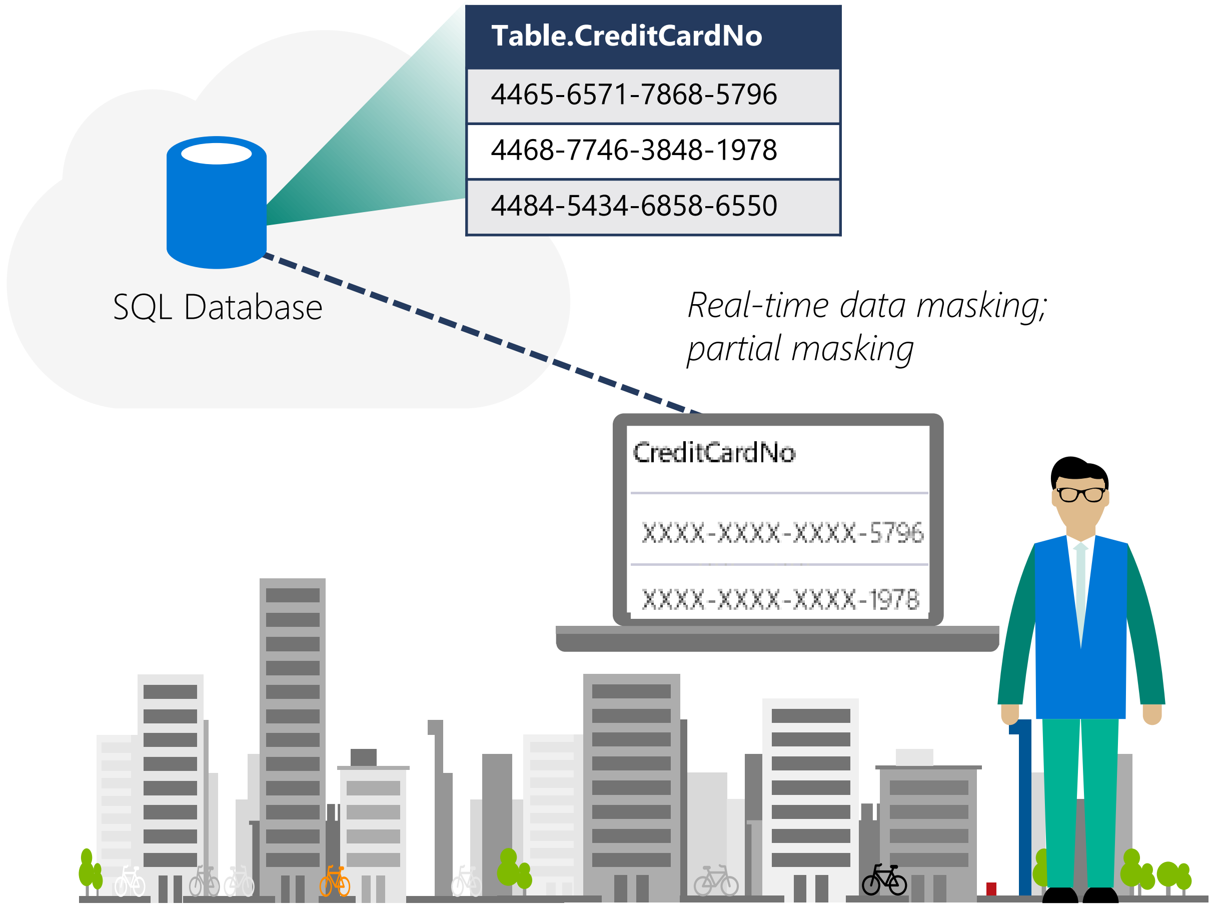


Table.CreditCardNo
4465-6571-7868-5796
4468-7746-3848-1978
4484-5434-6858-6550

CreditCardNo
XXXX-XXXX-XXXX-5796
XXXX-XXXX-XXXX-1978

Dynamic Data Masking scenarios

Developers can troubleshoot production data without viewing sensitive information.

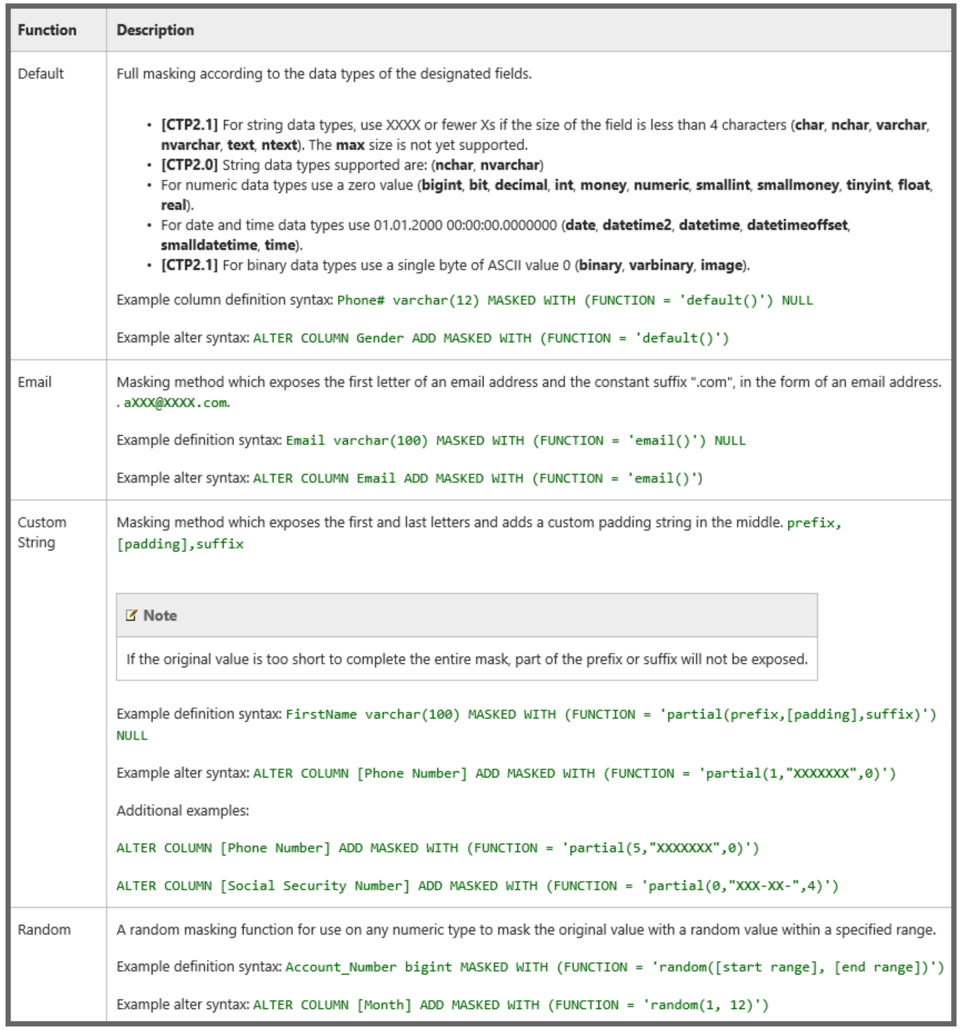
Customer Service representatives can view parts of sensitive data like credit card information.

Reports can be distributed with sensitive data obfuscated at the data layer.

Defining Dynamic Data Masking

Masking rule may be defined on column to protect data

Four types of masks are available



Function	Description
Default	Full masking according to the data types of the designated fields. • **[CTP2.1]** For string data types, use XXXX or fewer Xs if the size of the field is less than 4 characters (**char**, **nchar**, **varchar**, **nvarchar**, **text**, **ntext**). The **max** size is not yet supported. • **[CTP2.0]** String data types supported are: (**nchar**, **nvarchar**) • For numeric data types use a zero value (**bigint**, **bit**, **decimal**, **int**, **money**, **numeric**, **smallint**, **smallmoney**, **tinyint**, **float**, **real**). • For date and time data types use 01.01.2000 00:00:00.0000000 (**date**, **datetime2**, **datetime**, **datetimeoffset**, **smalldatetime**, **time**). • **[CTP2.1]** For binary data types use a single byte of ASCII value 0 (**binary**, **varbinary**, **image**). Example column definition syntax: `Phone# varchar(12) MASKED WITH (FUNCTION = 'default()') NULL` Example alter syntax: `ALTER COLUMN Gender ADD MASKED WITH (FUNCTION = 'default()')`
Email	Masking method which exposes the first letter of an email address and the constant suffix ".com", in the form of an email address. . `aXXX@XXXX.com`. Example definition syntax: `Email varchar(100) MASKED WITH (FUNCTION = 'email()') NULL` Example alter syntax: `ALTER COLUMN Email ADD MASKED WITH (FUNCTION = 'email()')`

Custom String	Masking method which exposes the first and last letters and adds a custom padding string in the middle. `prefix,[padding],suffix`
	Note
	If the original value is too short to complete the entire mask, part of the prefix or suffix will not be exposed.
	Example definition syntax: `FirstName varchar(100) MASKED WITH (FUNCTION = 'partial(prefix,[padding],suffix)') NULL`
	Example alter syntax: `ALTER COLUMN [Phone Number] ADD MASKED WITH (FUNCTION = 'partial(1,"XXXXXXX",0)')`
	Additional examples:
	`ALTER COLUMN [Phone Number] ADD MASKED WITH (FUNCTION = 'partial(5,"XXXXXXX",0)')`
	`ALTER COLUMN [Social Security Number] ADD MASKED WITH (FUNCTION = 'partial(0,"XXX-XX-",4)')`

Random	A random masking function for use on any numeric type to mask the original value with a random value within a specified range. Example definition syntax: `Account_Number bigint MASKED WITH (FUNCTION = 'random([start range], [end range])')` Example alter syntax: `ALTER COLUMN [Month] ADD MASKED WITH (FUNCTION = 'random(1, 12)')`

Dynamic Data Masking Functions

DEFAULT

RANDOM

CUSTOM

EMAIL

Default Data Masking Function

DEFAULT

Masking is according to the data types of the specified column.

- For string data types, uses XXXX
- For numeric data types use a zero value.
- For date and time data types use 01.01.1900 00:00:00.0000000

Random Data Masking Function

RANDOM

The Random Data Masking function is only applied on numeric data types. It displays a random value for the specified range.

- **Syntax**: Random([start], [end])

- **Actual syntax:** MASKED WITH (FUNCTION = ‘Random(1, 12)')

Custom Masking Function

CUSTOM

The Custom masking function allows the ability to create a custom mask using the Partial function.

- **Syntax**: Partial(prefix,[padding],suffix)

- **Prefix –** Starting characters to display.
- **Padding** –Custom string for masking.
- **Suffix –** Last characters to be displayed.

Email Data Masking Function

EMAIL

Masking will display the first character of an email address and mask the rest of the address with XXX@XXXX and will use the .com email suffix.

- The email address of Jane.Smith@AdventureWorks.com will be masked as JXXX@XXXX.com
- The email address of Susan.Jones@Contoso.net will be masked as SXXX@XXXX.com

Benefits of Dynamic Data Masking

Regulatory compliance

- Strong demand for applications to meet **privacy standards** recommended by regulating authorities

Sensitive data protection

- Enhanced protection against unauthorized access to sensitive data in application, and against exposure to developers or DBAs who need access to production database

Agility and transparency

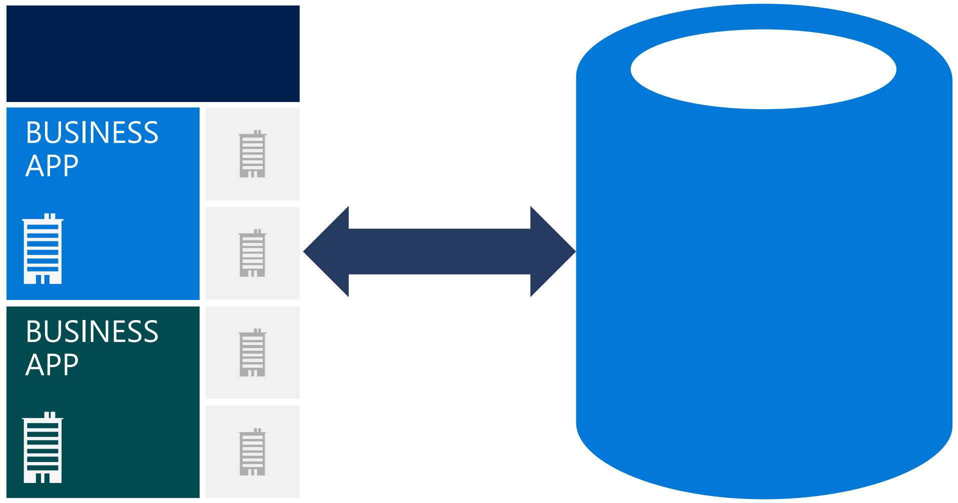
- Data is masked anytime, anywhere, with underlying data in database remaining intact
- Transparent to application and applied according to user privilege maintenance and complexity

Simplified Policies

- Limit access to sensitive data by defining policies to obfuscate specific database fields

Dynamic Data Masking Walkthrough

1) Security officer defines dynamic data-masking policy in T-SQL!
2) App user selects from employee table
3) Dynamic data-masking policy obfuscates sensitive data in query results

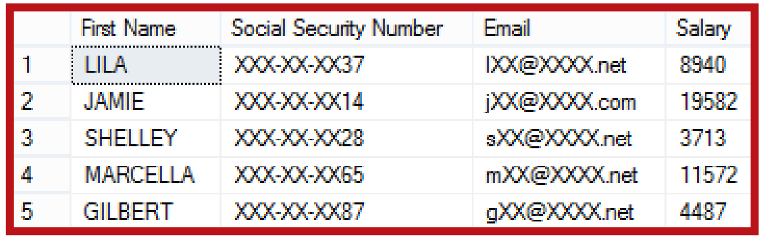


```
ALTER TABLE [Employee] ALTER COLUMN
[SocialSecurityNumber]
ADD MASKED WITH (FUNCTION = ‘SSN()’)

ALTER TABLE [Employee] ALTER COLUMN [Email]
ADD MASKED WITH (FUNCTION = ‘EMAIL()’)

ALTER TABLE [Employee] ALTER COLUMN [Salary]
ADD MASKED WITH (FUNCTION =
‘RANDOM(1,20000)’)

GRANT UNMASK to admin1
```

Security Officer

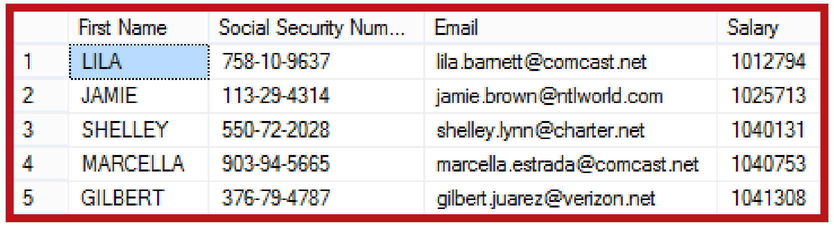
```
SELECT [Name],
[SocialSecurityNumber],
       [Email],
       [Salary]
FROM [Employee]
```

other login



	First Name	Social Security Number	Email	Salary
1	LILA	XXX-XX-XX37	lXX@XXXX.net	8940
2	JAMIE	XXX-XX-XX14	jXX@XXXX.com	19582
3	SHELLEY	XXX-XX-XX28	sXX@XXXX.net	3713
4	MARCELLA	XXX-XX-XX65	mXX@XXXX.net	11572
5	GILBERT	XXX-XX-XX87	gXX@XXXX.net	4487

admin1 login



	First Name	Social Security Num...	Email	Salary
1	LILA	758-10-9637	lila.barnett@comcast.net	1012794
2	JAMIE	113-29-4314	jamie.brown@ntlworld.com	1025713
3	SHELLEY	550-72-2028	shelley.lynn@charter.net	1040131
4	MARCELLA	903-94-5665	marcella.estrada@comcast.net	1040753
5	GILBERT	376-79-4787	gilbert.juarez@verizon.net	1041308

Querying for masked columns

Use sys.masked_columns view to query for table columns that have masking function

This view inherits from sys.columns, returning all columns in this view, plus is_masked and masking_function columns

This view only shows columns on which there is masking function applied.

```
SELECT c.name, tbl.name as table_name, c.is_masked, c.masking_function
FROM sys.masked_columns AS c
JOIN sys.tables AS tbl
    ON c.[object_id] = tbl.[object_id]
WHERE is_masked = 1;
```

Limitations and Restrictions

Masking rule cannot be defined for the following column types:

- Encrypted columns (Always Encrypted)
- FILESTREAM
- COLUMN_SET

For users without UNMASK permission, deprecated READTEXT, UPDATETEXT, and WRITETEXT statements do not function properly

Dynamic Data Masking Common Use Cases

SELECT INTO or INSERT INTO

Dynamic Data Masking is applied when running SQL Server Import and Export

Backing up databases with masked columns results in backups with masked data (for a user without UNMASK privileges)

Imported database will contain statically masked data copy data from a masked column results in masked data in target table

Demonstration

Dynamic Data Masking

- Implementing and demonstrating Dynamic Data Masking

Questions?

Microsoft