Amazon DynamoDB Encryption Client

Developer Guide

Amazon DynamoDB Encryption Client: Developer Guide

Table of Contents

What is the Amazon DynamoDB Encryption Client? ... 1
- Developed in open-source repositories ... 2
- Support and maintenance ... 2
- Sending feedback ... 2
- Which fields are encrypted and signed? ... 2
  - Encrypting attribute values ... 3
  - Signing the item ... 4
  - An encrypted and signed item ... 4
- How it works ... 5
- Client-side and server-side encryption ... 6
- Concepts ... 8
  - Cryptographic materials provider (CMP) ... 8
  - Item encryptors ... 9
  - Attribute actions ... 9
  - Material description ... 9
  - DynamoDB encryption context ... 10
  - Provider store ... 10

How to choose a cryptographic materials provider ... 12
- Direct KMS Provider ... 13
  - How to use it ... 14
  - How it works ... 15
- Wrapped Provider ... 18
  - How to use it ... 19
  - How it works ... 19
- Most Recent Provider ... 21
  - How to use it ... 22
  - How it works ... 23
  - Updates to the Most Recent Provider ... 28
- Static Provider ... 29
  - How to use it ... 30
  - How it works ... 30

Programming languages ... 32
- Java ... 32
  - Prerequisites ... 32
  - Installation ... 33
  - Using the DynamoDB Encryption Client for Java ... 33
  - Java examples ... 37
- Python ... 41
  - Prerequisites ... 42
  - Installation ... 42
  - Using the DynamoDB Encryption Client for Python ... 42
  - Python examples ... 44

Changing your data model ... 50
- Adding an attribute ... 50
- Removing an attribute ... 52

Troubleshooting ... 54
- Access denied ... 54
- Signature verification fails ... 55
- Issues with older version global tables ... 55
- Poor performance of the Most Recent Provider ... 56

Document history ... 57

What is the Amazon DynamoDB Encryption Client?

The Amazon DynamoDB Encryption Client is a software library that enables you to include client-side encryption in your Amazon DynamoDB design. The DynamoDB Encryption Client is designed to be implemented in new, unpopulated databases. Encrypting your sensitive data in transit and at rest helps ensure that your plaintext data isn't available to any third party, including AWS. The DynamoDB Encryption Client is provided free of charge under the Apache 2.0 license.

Important
The DynamoDB Encryption Client does not support the encryption of existing, unencrypted DynamoDB table data.

This developer guide provides a conceptual overview of the DynamoDB Encryption Client, including an introduction to its architecture (p. 5), details about how it protects DynamoDB table data (p. 2) and how it differs from DynamoDB server-side encryption (p. 6), guidance on selecting critical components for your application (p. 12), and examples in each programming language (p. 32) to help you get started.

The DynamoDB Encryption Client has the following benefits:

Designed especially for DynamoDB applications

You don't need to be a cryptography expert to use the DynamoDB Encryption Client. The implementations include helper methods that are designed to work with your existing DynamoDB applications.

After you create and configure the required components, the DynamoDB Encryption Client transparently encrypts and signs your table items when you add them to a table, and verifies and decrypts them when you retrieve them.

The DynamoDB Encryption Client supports most Amazon DynamoDB features, including global tables. However, you might need to make some configuration changes if you're using an older version of global tables. For details, see Issues with older version global tables (p. 55)..

Includes secure encryption and signing

The DynamoDB Encryption Client includes secure implementations that encrypt the attribute values in each table item using a unique encryption key, and then sign the item to protect it against unauthorized changes, such as adding or deleting attributes, or swapping encrypted values.

Uses cryptographic materials from any source

You can use the DynamoDB Encryption Client with encryption keys from any source, including your custom implementation or a cryptography service, such as AWS Key Management Service (AWS KMS) or AWS CloudHSM. The DynamoDB Encryption Client doesn't require an AWS account or any AWS service.

Programming language implementations are interoperable

The DynamoDB Encryption Client libraries are developed in open source projects on GitHub. They are currently available in Java and Python. All supported programming language implementations of the DynamoDB Encryption Client are interoperable. For example, you can encrypt data with the Java client and decrypt it with the Python client.

However, the DynamoDB Encryption Client is not compatible with the AWS Encryption SDK or the Amazon S3 Encryption Client. You cannot encrypt with one client-side library and decrypt with another.

Developed in open-source repositories

The Amazon DynamoDB Encryption Client is developed in open-source repositories on GitHub. You can use these repositories to view the code, read and submit issues, and find information that is specific to your language implementation.

- DynamoDB Encryption Client for Java — aws-dynamodb-encryption-java
- DynamoDB Encryption Client for Python — aws-dynamodb-encryption-python

Support and maintenance

The DynamoDB Encryption Client uses the same maintenance policy that the AWS SDK and Tools use, including its versioning and life-cycle phases. As a best practice, we recommend that you use the latest available version of the DynamoDB Encryption Client for your programming language, and upgrade as new versions are released.

Each programming language implementation of the DynamoDB Encryption Client is developed in a separate open-source GitHub repository. The life-cycle and support phase of each version is likely to vary among repositories. For example, a given version of the DynamoDB Encryption Client might be in the general availability (full support) phase in one programming language, but the end-of-support phase in a different programming language. We recommend that you use a fully supported version whenever possible and avoid versions that are no longer supported.

To find the life-cycle phase of DynamoDB Encryption Client versions for your programming language, see the `SUPPORT_POLICY.rst` file in each DynamoDB Encryption Client repository.

- DynamoDB Encryption Client for Java — SUPPORT_POLICY.rst
- DynamoDB Encryption Client for Python — SUPPORT_POLICY.rst

For more information, see the AWS SDKs and Tools maintenance policy in the AWS SDKs and Tools Reference Guide.

Sending feedback

We welcome your feedback! If you have a question or comment, or an issue to report, please use the following resources.

- If you discover a potential security vulnerability in the DynamoDB Encryption Client, please notify AWS security. Do not create a public GitHub issue.
- To provide feedback on the DynamoDB Encryption Client, file an issue in the aws-dynamodb-encryption-java or aws-dynamodb-encryption-python GitHub repository.
- To provide feedback on this documentation, use the feedback link on any page. You can also file an issue or contribute to aws-dynamodb-encryption-docs, the open-source repository for this documentation on GitHub.

Which fields are encrypted and signed?

In DynamoDB, a table is a collection of items. Each *item* is a collection of *attributes*. Each attribute has a name and a value.

The DynamoDB Encryption Client encrypts the values of attributes. Then, it calculates a signature over the attributes. You can specify which attribute values to encrypt and which to include in the signature. However, the DynamoDB Encryption Client is designed to be implemented in new, unpopulated databases. You need to add the encryption features to your DynamoDB applications before you send any data to DynamoDB.

Encryption protects the confidentiality of the attribute value. Signing provides integrity of all signed attributes and their relationship to each other, and provides authentication. It enables you to detect unauthorized changes to the item as a whole, including adding or deleting attributes, or substituting one encrypted value for another.

In an encrypted item, some data remains in plaintext, including the table name, all attribute names, the attribute values that you don't encrypt, and the names and values of the primary key (partition key and sort key) attributes. Do not store sensitive data in these fields.

Topics

- Encrypting attribute values (p. 3)
- Signing the item (p. 4)
- An encrypted and signed item (p. 4)

Encrypting attribute values

The DynamoDB Encryption Client encrypts the values (but not the names) of the attributes that you specify. To determine which attribute values are encrypted, use attribute actions (p. 9).

For example, this item includes `example` and `test` attributes.

```
'example': 'data',
'test': 'test-value',
...
```

If you encrypt the `example` attribute, but don't encrypt the `test` attribute, the results look like the following. The encrypted `example` attribute value is binary data, instead of a string.

```
'example': Binary(b"'b\x933\x9a+s\xf1\xd6a\xc5\xd5\x1aZ\xed\xd6\xce\xe9X\xf0T\xcb\x9fY\x9f
\xf3\xc9C\x83\r\xbb\\"),
'test': 'test-value'
...
```

The primary key attributes—partition key and sort key—of each item must remain in plaintext because DynamoDB uses them to find the item in the table. They should be signed, but not encrypted.

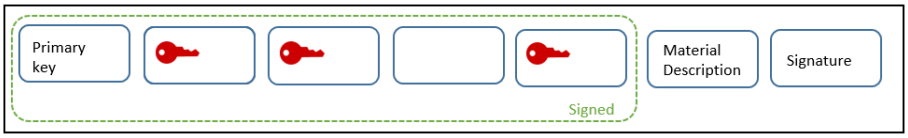
> **Warning**
> Do not encrypt the primary key attributes. They must remain in plaintext so DynamoDB can find the item without running a full table scan.

The helpers in each programming language identify the primary key attributes for you and ensure that their values are signed, but not encrypted. And, if you identify your primary key and then try to encrypt it, the client will throw an exception. If you need to encrypt the primary key for a special use case, use the lower-level item encryptor (p. 9) directly, but remember that DynamoDB will not be able to find your item without running a full table scan.

The DynamoDB Encryption Client also does not encrypt or sign the material description attribute (p. 9), which stores information that the DynamoDB Encryption Client needs to verify and decrypt the item.

Signing the item

After encrypting the specified attribute values, the DynamoDB Encryption Client calculates a digital signature over the names and values of attributes that you specify in the attribute actions (p. 9) object. The client saves the signature in an attribute that it adds to the item.



If you provide a table name, it's included in the signature. This allows you to detect that a signed item was moved to a different table, perhaps maliciously, such as moving an employee record from the `AllEmployees` to `TrustedEmployees` table. The DynamoDB Encryption Client gets the table name from the DynamoDB encryption context (p. 10), where it is an optional field.

Be sure to include the primary key in the signature. It's the default behavior when you use a helper. The signature captures the relationship between the primary key and other attributes in the item, and the signature validation verifies that the relationship hasn't changed.

The material description attribute (p. 9) is not encrypted or signed.

An encrypted and signed item

When the DynamoDB Encryption Client encrypts and signs a table item, the result is a standard DynamoDB table item with encrypted attribute values.

The following figure shows a part of an example encrypted and signed table item.

```
{
    '*amzn-ddb-map-desc*': Binary(b'\x00\x00\x00\x00\x00\x00\x00\x10amzn-ddb-env-alg\
\x00\x00\x00\xe0AQEBAHhA84wnXjEJdBbBBylRUFcZZK2j7xwh6UyLoL28nQ
+0FAAAAH4wfAYJKoZIhvcNAQcGoG8wbQIBADBoBgkqhkiG9w0BBwEwHgYJYIZIAWUDBAEuMBEEDPeFBydmoJD
izYl0R0C4M7wAK6E1/N/bgTmHI=\x00\x00\x00\x17amzn-ddb-map-signingAlg\x00\x00\x00\nHmacS
\x00\x00\x00\x11/CBC/PKCS5Padding\x00\x00\x00\x10amzn-ddb-sig-alg\x00\x00\x00\x0eHmac
\x00\x00\x00\x0faws-kms-ec-attr\x00\x00\x00\x06*keys*'),
    '*amzn-ddb-map-sig*': Binary(b"\xd3\xc6\xc7\n\xb7#\x13\xd1Y\xea\xe4.|^\xbd\xdf\xe
    'binary': Binary(b'!"\xc5\x92\xd7\x13\x1d\xe8Bs\x9b\x7f\xa8\x8e\x9c\xcf\x10\x1e\x
    'example': Binary(b"'b\x933\x9a+s\xf1\xd6a\xc5\xd5\x1aZ\xed\xd6\xce\xe9X\xf0T\xcb
    'numbers': Binary(b'\xd5\xa0\\d\xcc\x85\xf5\x1e\xb9-f!\xb9\xb8\x8a\x1aT\xbaq\xf7\
    'partition_attribute': 'value1',
    'sort_attribute': 55,
    'test': 'test-value'
}
```

The figure shows the following characteristics of table items that the DynamoDB Encryption Client encrypts and signs:

- All attribute names are in plaintext.
- The values of the primary key attributes are in plaintext. In this example, the partition key name is `partition_attribute` and the sort key name is `sort_attribute`.
- The values of any attributes that you tell the client not to encrypt remain in plaintext. In this example, the value of the `test` attribute is in plaintext.
- The values of encrypted attributes are binary data.
- The client adds a signature attribute (`*amzn-ddb-map-sig*`) to the item. Its value is the item signature.

- The client adds a material description attribute (p. 9) (`*amzn-ddb-map-desc*`) to the item. Its value describes how the attribute was encrypted and signed. The client uses this information to verify and decrypt the item. The material description attribute is not encrypted or signed.

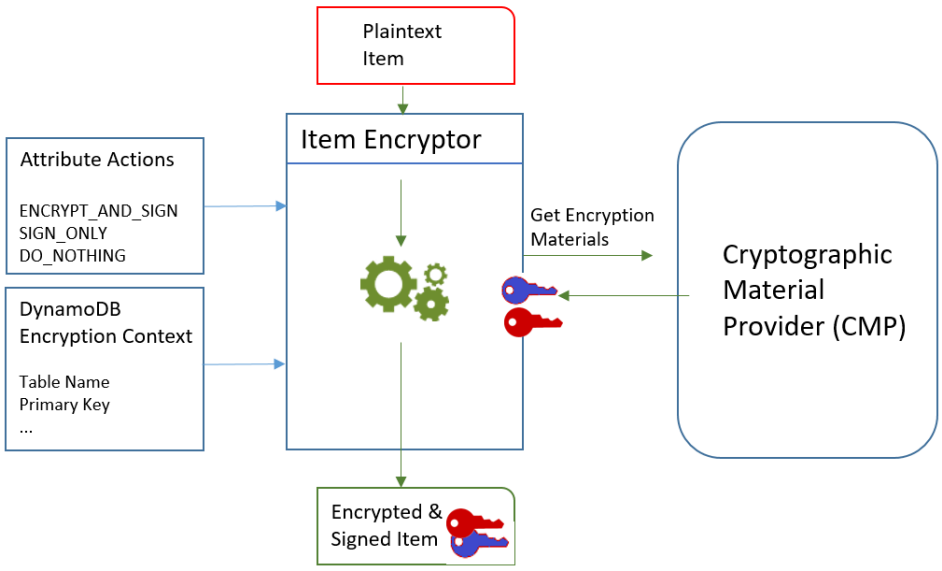
How the DynamoDB Encryption Client works

The DynamoDB Encryption Client is designed specifically to protect the data that you store in DynamoDB. The libraries include secure implementations that you can extend or use unchanged. And, most elements are represented by abstract elements so you can create and use compatible custom components.

Encrypting and signing table items

At the core of the DynamoDB Encryption Client is an *item encryptor* that encrypts, signs, verifies, and decrypts table items. It takes in information about your table items and instructions about which items to encrypt and sign. It gets the encryption materials, and instructions on how to use them, from a cryptographic material provider (p. 8) that you select and configure.

The following diagram shows a high-level view of this process.



To encrypt and sign a table item, the DynamoDB Encryption Client needs:

- **Information about the table.** It gets information about the table from a DynamoDB encryption context (p. 10) that you supply. Some helpers get the required information from DynamoDB and create the DynamoDB encryption context for you.

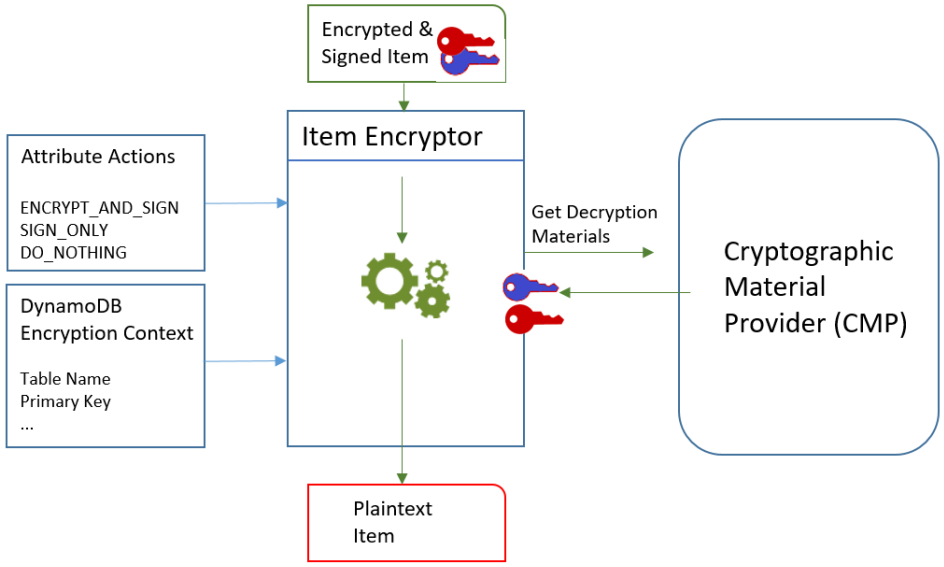
  Note
  The *DynamoDB encryption context* in the DynamoDB Encryption Client is not related to the *encryption context* in AWS Key Management Service (AWS KMS) and the AWS Encryption SDK.
- **Which attributes to encrypt and sign.** It gets this information from the attribute actions (p. 9) that you supply.
- **Encryption materials, including encryption and signing keys.** It gets these from a cryptographic materials provider (p. 8) (CMP) that you select and configure.
- **Instructions for encrypting and signing the item**. The CMP adds instructions for using the encryption materials, including encryption and signing algorithms, to the actual material description (p. 9).

The item encryptor (p. 9) uses all of these elements to encrypt and sign the item. The item encryptor also adds two attributes to the item: a material description attribute (p. 9) that contains the encryption and signing instructions (the actual material description), and an attribute that contains the signature. You can interact with the item encryptor directly, or use helper features that interact with the item encryptor for you to implement secure default behavior.

The result is a DynamoDB item containing encrypted and signed data.

Verifying and decrypting table items

These components also work together to verify and decrypt your item, as shown in the following diagram.



To verify and decrypt an item, the DynamoDB Encryption Client needs the same components, components with the same configuration, or components especially designed for decrypting the items, as follows:

- **Information about the table** from the DynamoDB encryption context (p. 10).
- **Which attributes to verify and decrypt.** It gets these from the attribute actions (p. 9).
- **Decryption materials, including verification and decryption keys**, from the cryptographic materials provider (p. 8) (CMP) that you select and configure.

  The encrypted item doesn't include any record of the CMP that was used to encrypt it. You must supply the same CMP, a CMP with the same configuration, or a CMP that is designed to decrypt items.
- **Information about how the item was encrypted and signed**, including the encryption and signing algorithms. The client gets these from the material description attribute (p. 9) in the item.

The item encryptor (p. 9) uses all of these elements to verify and decrypt the item. It also removes the material description and signature attributes. The result is a plaintext DynamoDB item.

Client-side and server-side encryption

The DynamoDB Encryption Client supports *client-side encryption*, where you encrypt your table data before you send it to DynamoDB. However, DynamoDB provides a server-side *encryption at rest* feature

that transparently encrypts your table when it is persisted to disk and decrypts it when you access the table.

The tools that you choose depend on the sensitivity of your data and the security requirements of your application. You can use both the DynamoDB Encryption Client and encryption at rest. When you send encrypted and signed items to DynamoDB, DynamoDB doesn't recognize the items as being protected. It just detects typical table items with binary attribute values.

Server-side encryption at rest

DynamoDB supports encryption at rest, a *server-side encryption* feature in which DynamoDB transparently encrypts your tables for you when the table is persisted to disk, and decrypts them when you access the table data.

With server-side encryption, your data is encrypted in transit over an HTTPS connection, decrypted at the DynamoDB endpoint, and then re-encrypted before being stored in DynamoDB.

- **Encryption by default.** DynamoDB transparently encrypts and decrypts all tables when they are written to disk. There is no option to enable or disable encryption at rest.
- **DynamoDB creates and manages the cryptographic keys.** The unique key for each table is protected by an AWS KMS key that never leaves AWS Key Management Service (AWS KMS) unencrypted. By default, DynamoDB uses an AWS owned key in the DynamoDB service account, but you can choose an AWS managed key or customer managed key in your account to protect some or all of your tables.
- **All table data is encrypted on disk.** When an encrypted table is saved to disk, DynamoDB encrypts all table data, including the primary key and local and global secondary indexes. If your table has a sort key, some of the sort keys that mark range boundaries are stored in plaintext in the table metadata.
- **Objects related to tables are encrypted, too.** Encryption at rest protects DynamoDB streams, global tables, and backups whenever they are written to durable media.
- **Your items are decrypted when you access them.** When you access the table, DynamoDB decrypts the part of the table that includes your target item, and returns the plaintext item to you.

DynamoDB Encryption Client

Client-side encryption provides end-to-end protection for your data, in transit and at rest, from its source to storage in DynamoDB. Your plaintext data is never exposed to any third party, including AWS. However, the DynamoDB Encryption Client is designed to be implemented in new, unpopulated databases. You need to add the encryption features to your DynamoDB applications before you send any data to DynamoDB.

- **Your data is protected in transit and at rest.** It is never exposed to any third party, including AWS.
- **You can sign your table Items.** You can direct the DynamoDB Encryption Client to calculate a signature over all or part of a table item, including the primary key attributes and the table name. This signature allows you to detect unauthorized changes to the item as a whole, including adding or deleting attributes, or swapping attribute values.
- **You choose how your cryptographic keys are generated and protected.** You can create and manage your keys, or use a cryptographic service, such as AWS Key Management Service or AWS CloudHSM, to generate and protect your keys.
- **You determine how your data is protected** by selecting a cryptographic materials provider (p. 12) (CMP), or writing one of your own. The CMP determines the encryption strategy used, including when unique keys are generated, and the encryption and signing algorithms that are used.
- **The DynamoDB Encryption Client doesn't encrypt the entire table.** You can encrypt selected items in a table, or selected attribute values in some or all items. However, the DynamoDB Encryption Client does not encrypt an entire item. It does not encrypt attribute names, or the names or values of the

primary key (partition key and sort key) attributes. For details about what is encrypted (and what is not), see Which fields are encrypted and signed? (p. 2).

AWS Encryption SDK

If you are encrypting data that you store in DynamoDB, we recommend the DynamoDB Encryption Client.

The AWS Encryption SDK is a client-side encryption library that helps you to encrypt and decrypt generic data. Although it can protect any type of data, it isn't designed to work with structured data, like database records. Unlike the DynamoDB Encryption Client, the AWS Encryption SDK cannot provide item-level integrity checking and it has no logic to recognize attributes or prevent encryption of primary keys.

If you use the AWS Encryption SDK to encrypt any element of your table, remember that it isn't compatible with the DynamoDB Encryption Client. You cannot encrypt with one library and decrypt with the other.

Amazon DynamoDB Encryption Client concepts

This topic explains the concepts and terminology used in the Amazon DynamoDB Encryption Client.

To learn how the components of the DynamoDB Encryption Client interact, see How the DynamoDB Encryption Client works (p. 5).

Topics

- Cryptographic materials provider (CMP) (p. 8)
- Item encryptors (p. 9)
- Attribute actions (p. 9)
- Material description (p. 9)
- DynamoDB encryption context (p. 10)
- Provider store (p. 10)

Cryptographic materials provider (CMP)

When implementing the DynamoDB Encryption Client, one of your first tasks is to select a cryptographic materials provider (p. 12) (CMP) (also known as an *encryption materials provider*). Your choice determines much of the rest of the implementation.

A *cryptographic materials provider* (CMP) collects, assembles, and returns the cryptographic materials that the item encryptor (p. 9) uses to encrypt and sign your table items. The CMP determines the encryption algorithms to use and how to generate and protect encryption and signing keys.

The CMP interacts with the item encryptor. The item encryptor requests encryption or decryption materials from the CMP, and the CMP returns them to the item encryptor. Then, the item encryptor uses the cryptographic materials to encrypt and sign, or verify and decrypt, the item.

You specify the CMP when you configure the client. You can create a compatible custom CMP, or use one of the many CMPs in the library. Most CMPs are available for multiple programming languages.

Item encryptors

The *item encryptor* is a lower-level component that performs cryptographic operations for the DynamoDB Encryption Client. It requests cryptographic materials from a cryptographic materials provider (p. 8) (CMP), then uses the materials that the CMP returns to encrypt and sign, or verify and decrypt, your table item.

You can interact with the item encryptor directly or use the helpers that your library provides. For example, the DynamoDB Encryption Client for Java includes an `AttributeEncryptor` helper class that you can use with the `DynamoDBMapper`, instead of interacting directly with the `DynamoDBEncryptor` item encryptor. The Python library includes `EncryptedTable`, `EncryptedClient`, and `EncryptedResource` helper classes that interact with the item encryptor for you.

Attribute actions

Attribute actions tell the item encryptor which actions to perform on each attribute of the item.

The attribute action values can be one of the following:

- **Encrypt and sign** – Encrypt the attribute value. Include the attribute (name and value) in the item signature.
- **Sign only** – Include the attribute in the item signature.
- **Do nothing** – Do not encrypt or sign the attribute.

For any attribute that can store sensitive data, use **Encrypt and sign**. For primary key attributes (partition key and sort key), use **Sign only**. The material description attribute (p. 9) and the signature attribute are not signed or encrypted. You don't need to specify attribute actions for these attributes.

Choose your attribute actions carefully. When in doubt, use **Encrypt and sign**. Once you have used the DynamoDB Encryption Client to protect your table items, you cannot change the action for an attribute without risking a signature validation error. For details, see Changing your data model (p. 50).

> **Warning**
> Do not encrypt the primary key attributes. They must remain in plaintext so DynamoDB can find the item without running a full table scan.

If the DynamoDB encryption context (p. 10) identifies your primary key attributes, the client will throw an error if you try to encrypt them.

The technique that you use to specify the attribute actions is different for each programming language. It might also be specific to helper classes that you use.

For details, see the documentation for your programming language.

- Python (p. 44)
- Java (p. 34)

Material description

The *material description* for an encrypted table item consists of information, such as encryption algorithms, about how the table item is encrypted and signed. The cryptographic materials provider (p. 8) (CMP) records the material description as it assembles the cryptographic materials for encryption and signing. Later, when it needs to assemble cryptographic materials to verify and decrypt the item, it uses the material description as its guide.

In the DynamoDB Encryption Client, the material description refers to three related elements:

Requested material description

Some cryptographic materials providers (p. 8) (CMPs) let you specify advanced options, such as an encryption algorithm. To indicate your choices, you add name-value pairs to the material description property of the DynamoDB encryption context (p. 10) in your request to encrypt a table item. This element is known as the *requested material description*. The valid values in the requested material description are defined by the CMP that you choose.

Note
Because the material description can override secure default values, we recommend that you omit the requested material description unless you have a compelling reason to use it.

Actual material description

The material description that the cryptographic materials providers (p. 8) (CMPs) return is known as the *actual material description*. It describes the actual values that the CMP used when it assembled the cryptographic materials. It usually consists of the requested material description, if any, with additions and changes.

Material description attribute

The client saves the actual material description in the *material description attribute* of the encrypted item. The material description attribute name is `amzn-ddb-map-desc` and its value is the actual material description. The client uses the values in the material description attribute to verify and decrypt the item.

DynamoDB encryption context

The *DynamoDB encryption context* supplies information about the table and item to the cryptographic materials provider (p. 8) (CMP). In advanced implementations, the DynamoDB encryption context can include a requested material description (p. 9).

When you encrypt table items, the DynamoDB encryption context is cryptographically bound to the encrypted attribute values. When you decrypt, if the DynamoDB encryption context is not an exact, case-sensitive match for the DynamoDB encryption context that was used to encrypt, the decrypt operation fails. If you interact with the item encryptor (p. 9) directly, you must provide a DynamoDB encryption context when you call an encrypt or decrypt method. Most helpers create the DynamoDB encryption context for you.

Note
The *DynamoDB encryption context* in the DynamoDB Encryption Client is not related to the *encryption context* in AWS Key Management Service (AWS KMS) and the AWS Encryption SDK.

The DynamoDB encryption context can include the following fields. All fields and values are optional.

- Table name
- Partition key name
- Sort key name
- Attribute name-value pairs
- Requested material description (p. 9)

Provider store

A *provider store* is a component that returns cryptographic materials providers (p. 8) (CMPs). The provider store can create the CMPs or get them from another source, such as another provider store. The

provider store saves versions of the CMPs that it creates in persistent storage in which each stored CMP is identified by the material name of the requester and version number.

The Most Recent Provider (p. 21) in the DynamoDB Encryption Client gets its CMPs from a provider store, but you can use the provider store to supply CMPs to any component. Each Most Recent Provider is associated with one provider store, but a provider store can supply CMPs to many requesters across multiple hosts.

The provider store creates new versions of CMPs on demand, and returns new and existing versions. It also returns the latest version number for a given material name. This lets the requester know when the provider store has a new version of its CMP that it can request.

The DynamoDB Encryption Client includes a MetaStore (p. 24), which is a provider store that creates Wrapped CMPs with keys that are stored in DynamoDB and encrypted by using an internal DynamoDB Encryption Client.

Learn more:

- Provider store: Java, Python
- MetaStore: Java, Python

How to choose a cryptographic materials provider

One of the most important decisions you make when using the DynamoDB Encryption Client is selecting a cryptographic materials provider (p. 8) (CMP). The CMP assembles and returns cryptographic materials to the item encryptor. It also determines how encryption and signing keys are generated, whether new key materials are generated for each item or are reused, and the encryption and signing algorithms that are used.

You can choose a CMP from the implementations provided in the DynamoDB Encryption Client libraries or build a compatible custom CMP. Your CMP choice might also depend on the programming language (p. 32) that you use.

This topic describes the most common CMPs and offers some advice to help you choose the best one for your application.

Direct KMS Materials Provider

The Direct KMS Materials Provider protects your table items under an AWS KMS key that never leaves AWS Key Management Service (AWS KMS) unencrypted. Your application doesn't have to generate or manage any cryptographic materials. Because it uses the AWS KMS key to generate unique encryption and signing keys for each item, this provider calls AWS KMS every time it encrypts or decrypts an item.

If you use AWS KMS and one AWS KMS call per transaction is practical for your application, this provider is a good choice.

For details, see Direct KMS Materials Provider (p. 13).

Wrapped Materials Provider (Wrapped CMP)

The Wrapped Materials Provider (Wrapped CMP) lets you generate and manage your wrapping and signing keys outside of the DynamoDB Encryption Client.

The Wrapped CMP generates a unique encryption key for each item. Then it uses wrapping (or unwrapping) and signing keys that you supply. As such, you determine how the wrapping and signing keys are generated and whether they are unique to each item or are reused. The Wrapped CMP is a secure alternative to the Direct KMS Provider (p. 13) for applications that don't use AWS KMS and can safely manage cryptographic materials.

For details, see Wrapped Materials Provider (p. 18).

Most Recent Provider

The *Most Recent Provider* is a cryptographic materials provider (p. 8) (CMP) that is designed to work with a provider store (p. 10). It gets CMPs from the provider store, and gets the cryptographic materials that it returns from the CMPs. The Most Recent Provider typically uses each CMP to satisfy multiple requests for cryptographic materials, but you can use the features of the provider store to control the extent to which materials are reused, determine how often its CMP is rotated, and even change the type of CMP that is used without changing the Most Recent Provider.

You can use the Most Recent Provider with any compatible provider store. The DynamoDB Encryption Client includes a MetaStore, which is a provider store that returns Wrapped CMPs.

The Most Recent Provider is a good choice for applications that need to minimize calls to their cryptographic source, and applications that can reuse some cryptographic materials without

violating their security requirements. For example, it allows you to protect your cryptographic materials under an AWS KMS key in AWS Key Management Service (AWS KMS) without calling AWS KMS every time you encrypt or decrypt an item.

For details, see Most Recent Provider (p. 21).

Static Materials Provider

The Static Materials Provider is designed for testing, proof-of-concept demonstrations, and legacy compatibility. It doesn't generate any unique cryptographic materials for each item. It returns the same encryption and signing keys that you supply, and those keys are used directly to encrypt, decrypt, and sign your table items.

> **Note**
> The Asymmetric Static Provider in the Java library is not a static provider. It just supplies alternate constructors for the Wrapped CMP (p. 18). It is safe for production use, but you should use the Wrapped CMP directly whenever possible.

Topics

- Direct KMS Materials Provider (p. 13)
- Wrapped Materials Provider (p. 18)
- Most Recent Provider (p. 21)
- Static Materials Provider (p. 29)

Direct KMS Materials Provider

The *Direct KMS Materials Provider* (Direct KMS Provider) protects your table items under an AWS KMS key that never leaves AWS Key Management Service (AWS KMS) unencrypted. This cryptographic materials provider (p. 8) returns a unique encryption key and signing key for every table item. To do so, it calls AWS KMS every time you encrypt or decrypt an item.

If you're processing DynamoDB items at a high frequency and large scale, you might exceed the AWS KMS requests-per-second limits, causing processing delays. If you need to exceed a limit, create a case in the AWS Support Center. You might also consider using a cryptographic materials provider with limited key reuse, such as the Most Recent Provider (p. 21).

To use the Direct KMS Provider, the caller must have an AWS account, at least one AWS KMS key, and permission to call the GenerateDataKey and Decrypt operations on the AWS KMS key. The AWS KMS key must be a symmetric encryption key; the DynamoDB Encryption Client does not support asymmetric encryption. If you are using a DynamoDB global table, you might want to specify an AWS KMS multi-Region key. For details, see How to use it (p. 14).

> **Note**
> When you use the Direct KMS Provider, the names and values of your primary key attributes appear in plaintext in the AWS KMS encryption context and AWS CloudTrail logs of related AWS KMS operations. However, the DynamoDB Encryption Client never exposes the plaintext of any encrypted attribute values.

The Direct KMS Provider is one of several cryptographic materials providers (p. 8) (CMPs) that the DynamoDB Encryption Client supports. For information about the other CMPs, see How to choose a cryptographic materials provider (p. 12).

For example code, see:

- Java: AwsKmsEncryptedItem
- Python: aws-kms-encrypted-table, aws-kms-encrypted-item

Topics

- How to use it (p. 14)
- How it works (p. 15)

How to use it

To create a Direct KMS Provider, use the key ID parameter to specify a symmetric encryption KMS key in your account. The value of the key ID parameter can be the key ID, key ARN, alias name, or alias ARN of the AWS KMS key. For details about the key identifiers, see Key identifiers in the *AWS Key Management Service Developer Guide*.

The Direct KMS Provider requires a symmetric encryption KMS key. You cannot use an asymmetric KMS key. However, you can use a multi-Region KMS key, a KMS key with imported key material, or a KMS key in a custom key store. You must have kms:GenerateDataKey and kms:Decrypt permission on the KMS key. As such, you must use a customer managed key, not an AWS managed or AWS owned KMS key.

The DynamoDB Encryption Client for Python determines the Region for calling AWS KMS from the Region in the key ID parameter value, if it includes one. Otherwise, it uses the Region in the AWS KMS client, if you specify one, or the Region that you configure in the AWS SDK for Python (Boto3). For information about Region selection in Python, see Configuration in the AWS SDK for Python (Boto3) API Reference.

The DynamoDB Encryption Client for Java determines the Region for calling AWS KMS from the Region in the AWS KMS client, if the client you specify includes a Region. Otherwise, it uses the Region that you configure in the AWS SDK for Java. For information about Region selection in the AWS SDK for Java, see AWS Region selection in the AWS SDK for Java Developer Guide.

Java

```
// Replace the example key ARN and Region with valid values for your application
final String keyArn = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
final String region = 'us-west-2'

final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, keyArn);
```

Python

The following example uses the key ARN to specify the AWS KMS key. If your key identifier doesn't include an AWS Region, the DynamoDB Encryption Client gets the Region from the configured Botocore session, if there is one, or from Boto defaults.

```
# Replace the example key ID with a valid value
kms_key = 'arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=kms_key)
```

If you are using Amazon DynamoDB global tables, we recommend that you encrypt your data under an AWS KMS multi-Region key. Multi-Region keys are AWS KMS keys in different AWS Regions that can be used interchangeably because they have the same key ID and key material. For details, see Using multi-Region keys in the *AWS Key Management Service Developer Guide*.

Note
If you are using the global tables version 2017.11.29, you must set attribute actions so the reserved replication fields are not encrypted or signed. For details, see Issues with older version global tables (p. 55).

To use a multi-Region key with the DynamoDB Encryption Client, create a multi-Region key and replicate it into the Regions in which your application runs. Then configure the Direct KMS Provider to use the multi-Region key in the Region in which the DynamoDB Encryption Client calls AWS KMS.

The following example configures the DynamoDB Encryption Client to encrypt data in the US East (N. Virginia) (us-east-1) Region and decrypt it in the US West (Oregon) (us-west-2) Region using a multi-Region key.

Java

In this example, the DynamoDB Encryption Client gets the Region for calling AWS KMS from the Region in the AWS KMS client. The `keyArn` value identifies a multi-Region key in the same Region.

```
// Encrypt in us-east-1

// Replace the example key ARN and Region with valid values for your application
final String usEastKey = 'arn:aws:kms:us-east-1:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab'
final String region = 'us-east-1'

final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, usEastKey);
```

```
// Decrypt in us-west-2

// Replace the example key ARN and Region with valid values for your application
final String usWestKey = 'arn:aws:kms:us-west-2:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab'
final String region = 'us-west-2'

final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, usWestKey);
```

Python

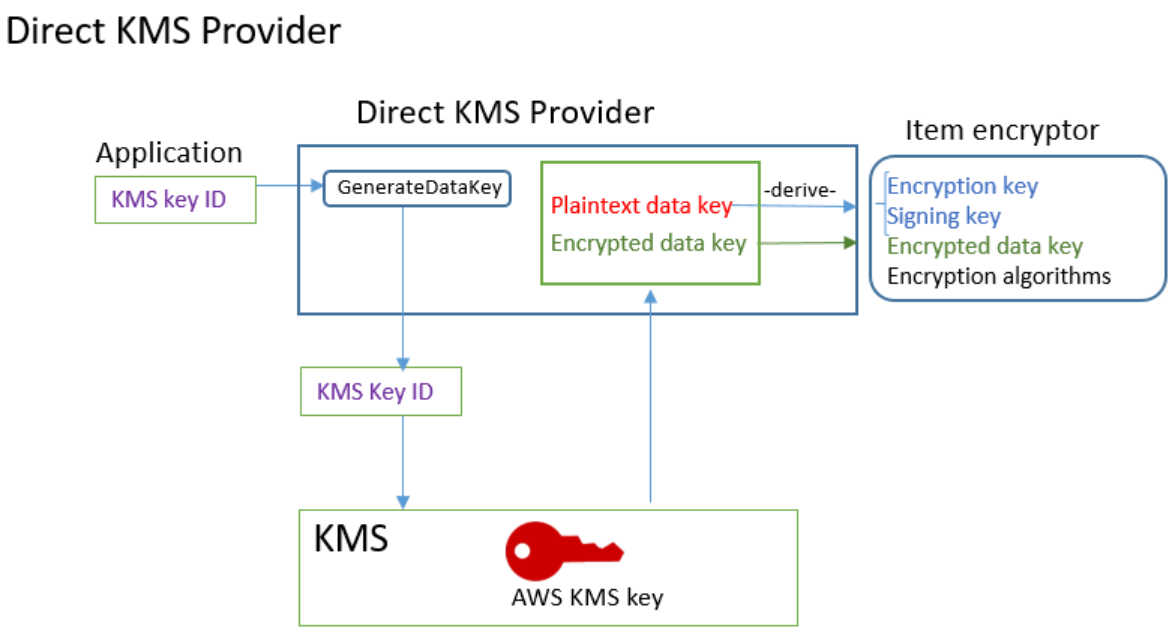
In this example, the DynamoDB Encryption Client gets the Region for calling AWS KMS from the Region in the key ARN.

```
# Encrypt in us-east-1

# Replace the example key ID with a valid value
us_east_key = 'arn:aws:kms:us-east-1:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=us_east_key)
```

```
# Decrypt in us-west-2

# Replace the example key ID with a valid value
us_west_key = 'arn:aws:kms:us-west-2:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=us_west_key)
```

How it works

The Direct KMS Provider returns encryption and signing keys that are protected by an AWS KMS key that you specify, as shown in the following diagram.



- To generate encryption materials, the Direct KMS Provider asks AWS KMS to generate a unique data key for each item using an AWS KMS key that you specify. It derives encryption and signing keys for the item from the plaintext copy of the data key, and then returns the encryption and signing keys, along with the encrypted data key, which is stored in the material description attribute (p. 9) of the item.

  The item encryptor uses the encryption and signing keys and removes them from memory as soon as possible. Only the encrypted copy of the data key from which they were derived is saved in the encrypted item.
- To generate decryption materials, the Direct KMS Provider asks AWS KMS to decrypt the encrypted data key. Then, it derives verification and signing keys from the plaintext data key, and returns them to the item encryptor.

  The item encryptor verifies the item and, if verification succeeds, decrypts the encrypted values. Then, it removes the keys from memory as soon as possible.

Get encryption materials

This section describes in detail the inputs, outputs, and processing of the Direct KMS Provider when it receives a request for encryption materials from the item encryptor (p. 9).

Input (from the application)

- The key ID of an AWS KMS key.

Input (from the item encryptor)

- DynamoDB encryption context (p. 10)

Output (to the item encryptor)

- Encryption key (plaintext)

- Signing key
- In actual material description (p. 9): These values are saved in the material description attribute that the client adds to the item.
  - amzn-ddb-env-key: Base64-encoded data key encrypted by the AWS KMS key
  - amzn-ddb-env-alg: Encryption algorithm, by default AES/256
  - amzn-ddb-sig-alg: Signing algorithm, by default, HmacSHA256/256
  - amzn-ddb-wrap-alg: kms

Processing

1. The Direct KMS Provider sends AWS KMS a request to use the specified AWS KMS key to generate a unique data key for the item. The operation returns a plaintext key and a copy that is encrypted under the AWS KMS key. This is known as the *initial key material*.

   The request includes the following values in plaintext in AWS KMS encryption context. These non-secret values are cryptographically bound to the encrypted object, so the same encryption context is required on decrypt. You can use these values to identify the call to AWS KMS in AWS CloudTrail logs.

   - amzn-ddb-env-alg – Encryption algorithm, by default AES/256
   - amzn-ddb-sig-alg – Signing algorithm, by default HmacSHA256/256
   - (Optional) aws-kms-table – *`table name`*
   - (Optional) *`partition key name`* – *`partition key value`* (binary values are Base64-encoded)
   - (Optional) *`sort key name`* – *`sort key value`* (binary values are Base64-encoded)

   The Direct KMS Provider gets the values for the AWS KMS encryption context from the DynamoDB encryption context (p. 10) for the item. If the DynamoDB encryption context doesn't include a value, such as the table name, that name-value pair is omitted from the AWS KMS encryption context.
2. The Direct KMS Provider derives a symmetric encryption key and a signing key from the data key. By default, it uses Secure Hash Algorithm (SHA) 256 and RFC5869 HMAC-based Key Derivation Function to derive a 256-bit AES symmetric encryption key and a 256-bit HMAC-SHA-256 signing key.
3. The Direct KMS Provider returns the output to the item encryptor.
4. The item encryptor uses the encryption key to encrypt the specified attributes and the signing key to sign them, using the algorithms specified in the actual material description. It removes the plaintext keys from memory as soon as possible.

Get decryption materials

This section describes in detail the inputs, outputs, and processing of the Direct KMS Provider when it receives a request for decryption materials from the item encryptor (p. 9).

Input (from the application)

- The key ID of an AWS KMS key.

  The value of the key ID can be the key ID, key ARN, alias name or alias ARN of the AWS KMS key. Any values that aren't included in the key ID, such as the Region, must be available in the AWS named profile. The key ARN provides all of the values that AWS KMS needs.

Input (from the item encryptor)

- A copy of the DynamoDB encryption context (p. 10) that contains the contents of the material description attribute.

Output (to the item encryptor)

- Encryption key (plaintext)
- Signing key

Processing

1. The Direct KMS Provider gets the encrypted data key from the material description attribute in the encrypted item.
2. It asks AWS KMS to use the specified AWS KMS key to decrypt the encrypted data key. The operation returns a plaintext key.

   This request must use the same AWS KMS encryption context that was used to generate and encrypt the data key.

   - aws-kms-table – *`table name`*
   - *`partition key name`* – *`partition key value`* (binary values are Base64-encoded)
   - (Optional) *`sort key name`* – *`sort key value`* (binary values are Base64-encoded)
   - amzn-ddb-env-alg – Encryption algorithm, by default AES/256
   - amzn-ddb-sig-alg – Signing algorithm, by default HmacSHA256/256
3. The Direct KMS Provider uses Secure Hash Algorithm (SHA) 256 and RFC5869 HMAC-based Key Derivation Function to derive a 256-bit AES symmetric encryption key and a 256-bit HMAC-SHA-256 signing key from the data key.
4. The Direct KMS Provider returns the output to the item encryptor.
5. The item encryptor uses the signing key to verify the item. If it succeeds, it uses the symmetric encryption key to decrypt the encrypted attribute values. These operations use the encryption and signing algorithms specified in the actual material description. The item encryptor removes the plaintext keys from memory as soon as possible.

Wrapped Materials Provider

The *Wrapped Materials Provider* (Wrapped CMP) lets you use wrapping and signing keys from any source with the DynamoDB Encryption Client. The Wrapped CMP does not depend on any AWS service. However, you must generate and manage your wrapping and signing keys outside of the client, including providing the correct keys to verify and decrypt the item.

The Wrapped CMP generates a unique item encryption key for each item. It wraps the item encryption key with the wrapping key that you provide and saves the wrapped item encryption key in the material description attribute (p. 9) of the item. Because you supply the wrapping and signing keys, you determine how the wrapping and signing keys are generated and whether they are unique to each item or are reused.

The Wrapped CMP is a secure implementation and a good choice for applications that can manage cryptographic materials.

The Wrapped CMP is one of several cryptographic materials providers (p. 8) (CMPs) that the DynamoDB Encryption Client supports. For information about the other CMPs, see How to choose a cryptographic materials provider (p. 12).

For example code, see:

- Java: AsymmetricEncryptedItem
- Python: wrapped-rsa-encrypted-table, wrapped-symmetric-encrypted-table

Topics

- How to use it (p. 19)
- How it works (p. 19)

How to use it

To create a Wrapped CMP, specify a wrapping key (required on encrypt), an unwrapping key (required on decrypt), and a signing key. You must supply keys when you encrypt and decrypt items.

The wrapping, unwrapping, and signing keys can be symmetric keys or asymmetric key pairs.

Java

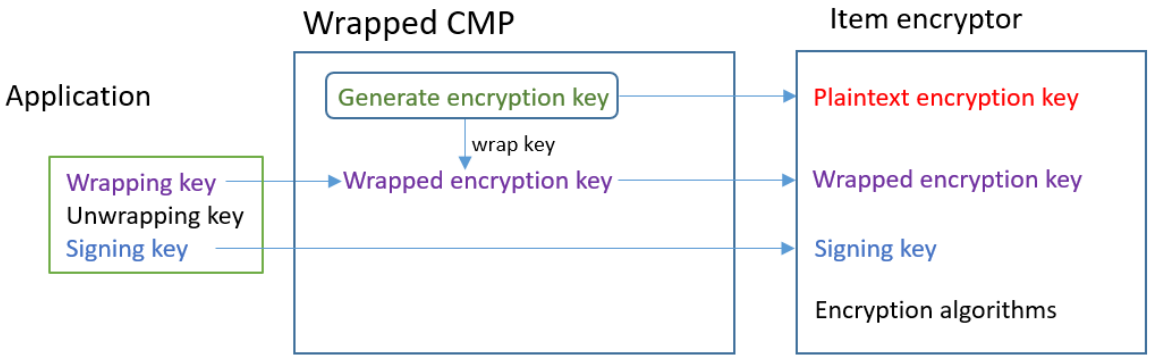
```
// This example uses asymmetric wrapping and signing key pairs
final KeyPair wrappingKeys = ...
final KeyPair signingKeys = ...

final WrappedMaterialsProvider cmp =
    new WrappedMaterialsProvider(wrappingKeys.getPublic(),
                                 wrappingKeys.getPrivate(),
                                 signingKeys);
```

Python

```
# This example uses symmetric wrapping and signing keys
wrapping_key = ...
signing_key  = ...

wrapped_cmp = WrappedCryptographicMaterialsProvider(
    wrapping_key=wrapping_key,
    unwrapping_key=wrapping_key,
    signing_key=signing_key
)
```

How it works

The Wrapped CMP generates a new item encryption key for every item. It uses the wrapping, unwrapping, and signing keys that you provide, as shown in the following diagram.



Get encryption materials

This section describes in detail the inputs, outputs, and processing of the Wrapped Materials Provider (Wrapped CMP) when it receives a request for encryption materials.

Input (from application)

- Wrapping key: An Advanced Encryption Standard (AES) symmetric key, or an RSA public key. Required if any attribute values are encrypted. Otherwise, it is optional and ignored.
- Unwrapping key: Optional and ignored.
- Signing key

Input (from the item encryptor)

- DynamoDB encryption context (p. 10)

Output (to the item encryptor):

- Plaintext item encryption key
- Signing key (unchanged)
- Actual material description (p. 9): These values are saved in the material description attribute (p. 9) that the client adds to the item.
  - `amzn-ddb-env-key`: Base64-encoded wrapped item encryption key
  - `amzn-ddb-env-alg`: Encryption algorithm used to encrypt the item. The default is AES-256-CBC.
  - `amzn-ddb-wrap-alg`: The wrapping algorithm that the Wrapped CMP used to wrap the item encryption key. If the wrapping key is an AES key, the key is wrapped using unpadded `AES-Keywrap` as defined in RFC 3394. If the wrapping key is an RSA key, the key is encrypted by using RSA OAEP with MGF1 padding.

Processing

When you encrypt an item, you pass in a wrapping key and a signing key. An unwrapping key is optional and ignored.

1. The Wrapped CMP generates a unique symmetric item encryption key for the table item.
2. It uses the wrapping key that you specify to wrap the item encryption key. Then, it removes it from memory as soon as possible.
3. It returns the plaintext item encryption key, the signing key that you supplied, and an actual material description (p. 9) that includes the wrapped item encryption key, and the encryption and wrapping algorithms.
4. The item encryptor uses the plaintext encryption key to encrypt the item. It uses the signing key that you supplied to sign the item. Then, it removes the plaintext keys from memory as soon as possible. It copies the fields in the actual material description, including the wrapped encryption key (`amzn-ddb-env-key`), to the material description attribute of the item.

Get decryption materials

This section describes in detail the inputs, outputs, and processing of the Wrapped Materials Provider (Wrapped CMP) when it receives a request for decryption materials.

Input (from application)

- Wrapping key: Optional and ignored.
- Unwrapping key: The same Advanced Encryption Standard (AES) symmetric key or RSA private key that corresponds to the RSA public key used to encrypt. Required if any attribute values are encrypted. Otherwise, it is optional and ignored.
- Signing key

Input (from the item encryptor)

- A copy of the DynamoDB encryption context (p. 10) that contains the contents of the material description attribute.

Output (to the item encryptor)

- Plaintext item encryption key
- Signing key (unchanged)

Processing

When you decrypt an item, you pass in an unwrapping key and a signing key. A wrapping key is optional and ignored.

1. The Wrapped CMP gets the wrapped item encryption key from the material description attribute of the item.
2. It uses the unwrapping key and algorithm to unwrap the item encryption key.
3. It returns the plaintext item encryption key, the signing key, and encryption and signing algorithms to the item encryptor.
4. The item encryptor uses the signing key to verify the item. If it succeeds, it uses the item encryption key to decrypt the item. Then, it removes the plaintext keys from memory as soon as possible.

Most Recent Provider

The *Most Recent Provider* is a cryptographic materials provider (p. 8) (CMP) that is designed to work with a provider store (p. 10). It gets CMPs from the provider store, and gets the cryptographic materials that it returns from the CMPs. It typically uses each CMP to satisfy multiple requests for cryptographic materials. But you can use the features of its provider store to control the extent to which materials are reused, determine how often its CMP is rotated, and even change the type of CMP that it uses without changing the Most Recent Provider.

> **Note**
> The code associated with the `MostRecentProvider` symbol for the Most Recent Provider might store cryptographic materials in memory for the lifetime of the process. It might allow a caller to use keys that they're no longer authorized to use.
> The `MostRecentProvider` symbol is deprecated in older supported versions of the DynamoDB Encryption Client and removed from version 2.0.0. It is replaced by the `CachingMostRecentProvider` symbol. For details, see Updates to the Most Recent Provider (p. 28).

The Most Recent Provider is a good choice for applications that need to minimize calls to the provider store and its cryptographic source, and applications that can reuse some cryptographic materials without violating their security requirements. For example, It allows you to protect your cryptographic materials under an AWS KMS key in AWS Key Management Service (AWS KMS) without calling AWS KMS every time you encrypt or decrypt an item.

The provider store that you choose determines the type of CMPs that the Most Recent Provider uses and how often it gets a new CMP. You can use any compatible provider store with the Most Recent Provider, including custom provider stores that you design.

The DynamoDB Encryption Client includes a *MetaStore* that creates and returns Wrapped Materials Providers (p. 18) (Wrapped CMPs). The MetaStore saves multiple versions of the Wrapped CMPs that it generates in an internal DynamoDB table and protects them with client-side encryption by an internal instance of the DynamoDB Encryption Client.

You can configure the MetaStore to use any type of internal CMP to protect the materials in the table, including a Direct KMS Provider (p. 13) that generates cryptographic materials protected by your AWS KMS key, a Wrapped CMP that uses wrapping and signing keys that you supply, or a compatible custom CMP that you design.

For example code, see:

- Java: MostRecentEncryptedItem
- Python: most_recent_provider_encrypted_table

Topics

- How to use it (p. 22)
- How it works (p. 23)
- Updates to the Most Recent Provider (p. 28)

How to use it

To create a Most Recent Provider, you need to create and configure a provider store, and then create a Most Recent Provider that uses the provider store.

The following examples show how to create a Most Recent Provider that uses a MetaStore and protects the versions in its internal DynamoDB table with cryptographic materials from a Direct KMS Provider (p. 13). These examples use the `CachingMostRecentProvider` (p. 28) symbol.

Each Most Recent Provider has a name that identifies its CMPs in the MetaStore table, a time-to-live (p. 25) (TTL) setting, and a cache size setting that determines how many entries the cache can hold. These examples set the cache size to 1000 entries and a TTL of 60 seconds.

Java

```
// Set the name for MetaStore's internal table
final String keyTableName = 'metaStoreTable'

// Set the Region and AWS KMS key
final String region = 'us-west-2'
final String keyArn = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'

// Set the TTL and cache size
final long ttlInMillis = 60000;
final long cacheSize = 1000;

// Name that identifies the MetaStore's CMPs in the provider store
final String materialName = 'testMRP'

// Create an internal DynamoDB client for the MetaStore
final AmazonDynamoDB ddb =
 AmazonDynamoDBClientBuilder.standard().withRegion(region).build();
```

```
// Create an internal Direct KMS Provider for the MetaStore
final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider kmsProv = new DirectKmsMaterialProvider(kms, keyArn);

// Create an item encryptor for the MetaStore,
// including the Direct KMS Provider
final DynamoDBEncryptor keyEncryptor = DynamoDBEncryptor.getInstance(kmsProv);

// Create the MetaStore
final MetaStore metaStore = new MetaStore(ddb, keyTableName, keyEncryptor);

//Create the Most Recent Provider
final CachingMostRecentProvider cmp = new CachingMostRecentProvider(metaStore,
 materialName, ttlInMillis, cacheSize);
```

Python

```
# Designate an AWS KMS key
kms_key_id = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'

# Set the name for MetaStore's internal table
meta_table_name = 'metaStoreTable'

# Name that identifies the MetaStore's CMPs in the provider store
material_name = 'testMRP'

# Create an internal DynamoDB table resource for the MetaStore
meta_table = boto3.resource('dynamodb').Table(meta_table_name)

# Create an internal Direct KMS Provider for the MetaStore
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=kms_key_id)

# Create the MetaStore with the Direct KMS Provider
meta_store = MetaStore(
    table=meta_table,
    materials_provider=kms_cmp
)

# Create a Most Recent Provider using the MetaStore
#    Sets the TTL (in seconds) and cache size (# entries)
most_recent_cmp = MostRecentProvider(
    provider_store=meta_store,
    material_name=material_name,
    version_ttl=60.0,
    cache_size=1000
)
```

How it works

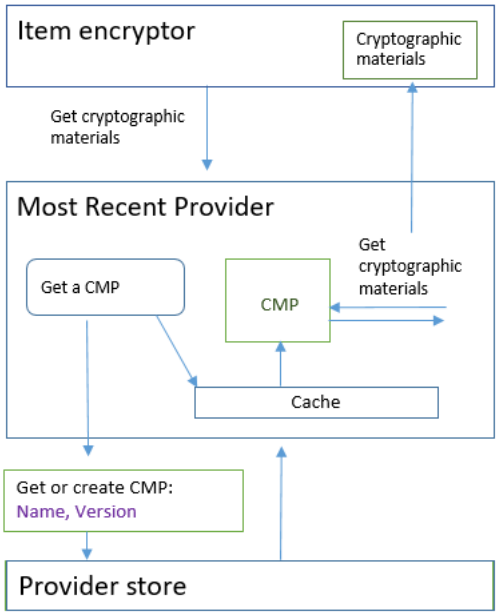
The Most Recent Provider gets CMPs from a provider store. Then, it uses the CMP to generate the cryptographic materials that it returns to the item encryptor.

About the Most Recent Provider

The Most Recent Provider gets a cryptographic materials provider (p. 8) (CMP) from a provider store (p. 10). Then, it uses the CMP to generate the cryptographic materials it returns. Each Most Recent Provider is associated with one provider store, but a provider store can supply CMPs to multiple providers across multiple hosts.

The Most Recent Provider can work with any compatible CMP from any provider store. It requests encryption or decryption materials from the CMP and returns the output to the item encryptor. It does not perform any cryptographic operations.

To request a CMP from its provider store, the Most Recent Provider supplies its material name and the version of an existing CMP it wants to use. For encryption materials, the Most Recent Provider always requests the maximum ("most recent") version. For decryption materials, it requests the version of the CMP that was used to create the encryption materials, as shown in the following diagram.



The Most Recent Provider saves versions of the CMPs that the provider store returns in a local Least Recently Used (LRU) cache in memory. The cache enables the Most Recent Provider to get the CMPs that it needs without calling the provider store for every item. You can clear the cache on demand.

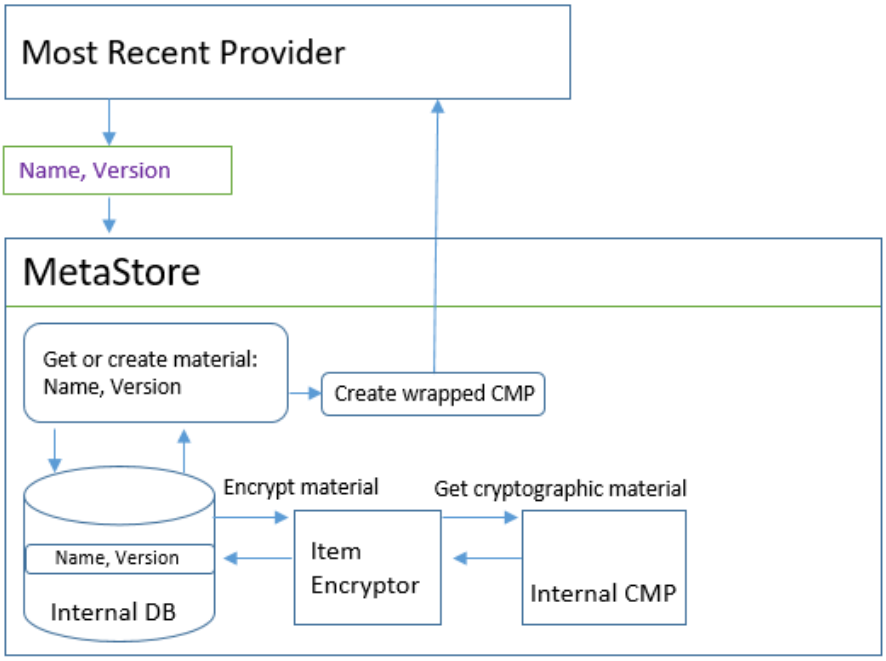
The Most Recent Provider uses a configurable time-to-live value (p. 25) that you can adjust based on the characteristics of your application.

About the MetaStore

You can use a Most Recent Provider with any provider store, including a compatible custom provider store. The DynamoDB Encryption Client includes a MetaStore, a secure implementation that you can configure and customize.

A *MetaStore* is a provider store (p. 10) that creates and returns Wrapped CMPs (p. 18) that are configured with the wrapping key, unwrapping key, and signing key that Wrapped CMPs require. A MetaStore is a secure option for a Most Recent Provider because Wrapped CMPs always generate unique item encryption keys for every item. Only the wrapping key that protects the item encryption key and the signing keys are reused.

The following diagram shows the components of the MetaStore and how it interacts with the Most Recent Provider.



The MetaStore generates the Wrapped CMPs, and then stores them (in encrypted form) in an internal DynamoDB table. The partition key is the name of the Most Recent Provider material; the sort key its version number. The materials in the table are protected by an internal DynamoDB Encryption Client, including an item encryptor and internal cryptographic materials provider (p. 8) (CMP).

You can use any type of internal CMP in your MetaStore, including the a Direct KMS Provider (p. 18), a Wrapped CMP with cryptographic materials that you provide, or a compatible custom CMP. If the internal CMP in your MetaStore is a Direct KMS Provider, your reusable wrapping and signing keys are protected under a AWS KMS key in AWS Key Management Service (AWS KMS). The MetaStore calls AWS KMS every time it adds a new CMP version to its internal table or gets a CMP version from its internal table.

Setting a time-to-live value

You can set a time-to-live (TTL) value for each Most Recent Provider that you create. In general, use the lowest TTL value that is practical for your application.

The use of the TTL value is changed in the `CachingMostRecentProvider` symbol for the Most Recent Provider.

> **Note**
> The `MostRecentProvider` symbol for the Most Recent Provider is deprecated in older supported versions of the DynamoDB Encryption Client and removed from version 2.0.0. It is replaced by the `CachingMostRecentProvider` symbol. We recommend that you update your code as soon as possible. For details, see Updates to the Most Recent Provider (p. 28).

`CachingMostRecentProvider`

The `CachingMostRecentProvider` uses the TTL value in two different ways.

- The TTL determines how often the Most Recent Provider checks the provider store for a new version of the CMP. If a new version is available, the Most Recent Provider replaces its CMP and refreshes its cryptographic materials. Otherwise, it continues to use its current CMP and cryptographic materials.
- The TTL determines how long CMPs in the cache can be used. Before it uses a cached CMP for encryption, the Most Recent Provider evaluates its time in the cache. If the CMP cache time

exceeds the TTL, the CMP is evicted from the cache and the Most Recent Provider gets a new, latest-version CMP from its provider store.

`MostRecentProvider`

In the `MostRecentProvider`, the TTL determines how often the Most Recent Provider checks the provider store for a new version of the CMP. If a new version is available, the Most Recent Provider replaces its CMP and refreshes its cryptographic materials. Otherwise, it continues to use its current CMP and cryptographic materials.

The TTL does not determine how often a new CMP version is created. You create new CMP versions by rotating the cryptographic materials (p. 26).

An ideal TTL value varies with the application and its latency and availability goals. A lower TTL improves your security profile by reducing the time that cryptographic materials are stored in memory. Also, a lower TTL refreshes critical information more frequently. For example, if your internal CMP is a Direct KMS Provider (p. 13), it verifies more frequently that the caller is still authorized to use an AWS KMS key.

However, if the TTL is too brief, the frequent calls to the provider store can increase your costs and cause your provider store to throttle requests from your application and other applications that share your service account. You might also benefit from coordinating the TTL with the rate at which you rotate cryptographic materials.

During testing, vary the TTL and cache size under different work loads until you find a configuration that works for your application and your security and performance standards.

Rotating cryptographic materials

When a Most Recent Provider needs encryption materials, it always uses the most recent version of its CMP that it knows about. The frequency that it checks for a newer version is determined by the time-to-live (p. 25) (TTL) value that you set when you configure the Most Recent Provider.

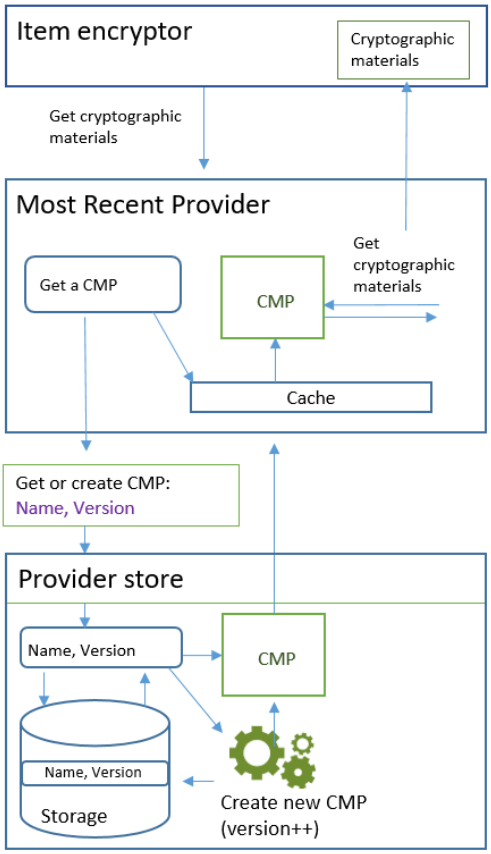
When the TTL expires, the Most Recent Provider checks the provider store for newer version of the CMP. If one is available, the Most Recent Provider get it and replaces the CMP in its cache. It uses this CMP and its cryptographic materials until it discovers that provider store has a newer version.

To tell the provider store to create a new version of a CMP for a Most Recent Provider, call the provider store's Create New Provider operation with the material name of the Most Recent Provider. The provider store creates a new CMP and saves an encrypted copy in its internal storage with a greater version number. (It also returns a CMP, but you can discard it.) As a result, the next time the Most Recent Provider queries the provider store for the maximum version number of its CMPs, it gets the new greater version number, and uses it in subsequent requests to the store to see if a new version of the CMP has been created.

You can schedule your Create New Provider calls based on time, the number of items or attributes processed, or any other metric that makes sense for your application.

Get encryption materials

The Most Recent Provider uses the following process, shown in this diagram, to get the encryption materials that it returns to the item encryptor. The output depends on the type of CMP that the provider store returns. The Most Recent Provider can use any compatible provider store, including the MetaStore that is included in the DynamoDB Encryption Client.



When you create a Most Recent Provider by using the `CachingMostRecentProvider` symbol (p. 28), you specify a provider store, a name for the Most Recent Provider, and a time-to-live (p. 25) (TTL) value. You can also optionally specify a cache size, which determines the maximum number of cryptographic materials that can exist in the cache.

When the item encryptor asks the Most Recent Provider for encryption materials, the Most Recent Provider begins by searching its cache for the latest version of its CMP.

- If it finds the latest version CMP in its cache and the CMP has not exceeded the TTL value, the Most Recent Provider uses the CMP to generate encryption materials. Then, it returns the encryption materials to the item encryptor. This operation does not require a call to the provider store.
- If the latest version of the CMP is not in its cache, or if it is in the cache but has exceeded its TTL value, the Most Recent Provider requests a CMP from its provider store. The request includes the Most Recent Provider material name and the maximum version number that it knows.
  1. The provider store returns a CMP from its persistent storage. If the provider store is a MetaStore, it gets an encrypted Wrapped CMP from its internal DynamoDB table by using the Most Recent Provider material name as the partition key and the version number as the sort key. The MetaStore uses its internal item encryptor and internal CMP to decrypt the Wrapped CMP. Then, it returns the plaintext CMP to the Most Recent Provider . If the internal CMP is a Direct KMS Provider (p. 13), this step includes a call to the AWS Key Management Service (AWS KMS).
  2. The CMP adds the `amzn-ddb-meta-id` field to the actual material description (p. 9). Its value is the material name and version of the CMP in its internal table. The provider store returns the CMP to the Most Recent Provider.
  3. The Most Recent Provider caches the CMP in memory.
  4. The Most Recent Provider uses the CMP to generate encryption materials. Then, it returns the encryption materials to the item encryptor.

Get decryption materials

When the item encryptor asks the Most Recent Provider for decryption materials, the Most Recent Provider uses the following process to get and return them.

1. The Most Recent Provider asks the provider store for the version number of the cryptographic materials that were used to encrypt the item. It passes in the actual material description from the material description attribute (p. 9) of the item.
2. The provider store gets the encrypting CMP version number from the `amzn-ddb-meta-id` field in the actual material description and returns it to the Most Recent Provider.
3. The Most Recent Provider searches its cache for the version of CMP that was used to encrypt and sign the item.

- If it finds the matching version of the CMP is in its cache and the CMP has not exceeded the time-to-live (TTL) value (p. 25), the Most Recent Provider uses the CMP to generate decryption materials. Then, it returns the decryption materials to the item encryptor. This operation does not require a call to the provider store or any other CMP.
- If the matching version of the CMP is not in its cache, or if the cached AWS KMS key has exceeded its TTL value, the Most Recent Provider requests a CMP from its provider store. It sends its material name and the encrypting CMP version number in the request.
  1. The provider store searches its persistent storage for the CMP by using the Most Recent Provider name as the partition key and the version number as the sort key.
     - If the name and version number are not in its persistent storage, the provider store throws an exception. If the provider store was used to generate the CMP, the CMP should be stored in its persistent storage, unless it was intentionally deleted.
     - If the CMP with the matching name and version number are in the provider store's persistent storage, the provider store returns the specified CMP to the Most Recent Provider.

       If the provider store is a MetaStore, it gets the encrypted CMP from its DynamoDB table. Then, it uses cryptographic materials from its internal CMP to decrypt the encrypted CMP before it returns the CMP to Most Recent Provider. If the internal CMP is a Direct KMS Provider (p. 13), this step includes a call to the AWS Key Management Service (AWS KMS).
  2. The Most Recent Provider caches the CMP in memory.
  3. The Most Recent Provider uses the CMP to generate decryption materials. Then, it returns the decryption materials to the item encryptor.

Updates to the Most Recent Provider

The symbol for the Most Recent Provider is changed from `MostRecentProvider` to `CachingMostRecentProvider`.

> **Note**
> The `MostRecentProvider` symbol, which represents the Most Recent Provider, is deprecated in version 1.15 of the DynamoDB Encryption Client for Java and version 1.3 of the DynamoDB Encryption Client for Python and removed from versions 2.0.0 of the DynamoDB Encryption Client in both language implementations. Instead, use the `CachingMostRecentProvider`.

The `CachingMostRecentProvider` implements the following changes:

- The `CachingMostRecentProvider` periodically removes cryptographic materials from memory when their time in memory exceeds the configured time-to-live (TTL) value (p. 25).

The `MostRecentProvider` might store cryptographic materials in memory for the lifetime of the process. As a result, the Most Recent Provider might not be aware of authorization changes. It might use encryption keys after the caller's permissions to use them are revoked.

If you can't update to this new version, you can get a similar effect by periodically calling the `clear()` method on the cache. This method manually flushes the cache contents and requires the Most Recent Provider to request a new CMP and new cryptographic materials.

- The `CachingMostRecentProvider` also includes a cache size setting that gives you more control over the cache.

To update to the `CachingMostRecentProvider`, you have to change the symbol name in your code. In all other respects, the `CachingMostRecentProvider` is fully backwards compatible with the `MostRecentProvider`. You don't need to re-encrypt any table items.

However, the `CachingMostRecentProvider` generates more calls to the underlying key infrastructure. It calls the provider store at least once in each time-to-live (TTL) interval. Applications with numerous active CMPs (due to frequent rotation) or applications with large fleets are most likely to be sensitive to this change.

Before releasing your updated code, test it thoroughly to ensure that the more frequent calls don't impair your application or cause throttling by services on which your provider depends, such as AWS Key Management Service (AWS KMS) or Amazon DynamoDB. To mitigate any performance problems, adjust the cache size and the time-to-live of the `CachingMostRecentProvider` based on the performance characteristics you observe. For guidance, see Setting a time-to-live value (p. 25).

Static Materials Provider

The *Static Materials Provider* (Static CMP) is a very simple cryptographic materials provider (p. 8) (CMP) that is intended for testing, proof-of-concept demonstrations, and legacy compatibility.

To use the Static CMP to encrypt a table item, you supply an Advanced Encryption Standard (AES) symmetric encryption key and a signing key or key pair. You must supply the same keys to decrypt the encrypted item. The Static CMP does not perform any cryptographic operations. Instead, it passes the encryption keys that you supply to the item encryptor unchanged. The item encryptor encrypts the items directly under the encryption key. Then, it uses the signing key directly to sign them.

Because the Static CMP does not generate any unique cryptographic materials, all table items that you process are encrypted with the same encryption key and signed by the same signing key. When you use the same key to encrypt the attributes values in numerous items or use the same key or key pair to sign all items, you risk exceeding the cryptographic limits of the keys.

> **Note**
> The Asymmetric Static Provider in the Java library is not a static provider. It just supplies alternate constructors for the Wrapped CMP (p. 18). It's safe for production use, but you should use the Wrapped CMP directly whenever possible.

The Static CMP is one of several cryptographic materials providers (p. 8) (CMPs) that the DynamoDB Encryption Client supports. For information about the other CMPs, see How to choose a cryptographic materials provider (p. 12).

For example code, see:

- Java: SymmetricEncryptedItem

Topics

- How to use it (p. 30)

- How it works (p. 30)

How to use it

To create a static provider, supply an encryption key or key pair and a signing key or key pair. You need to provide key material to encrypt and decrypt table items.

Java

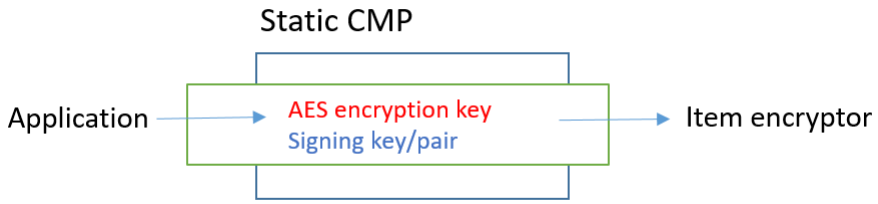
```
// To encrypt
SecretKey cek = ...;        // Encryption key
SecretKey macKey =  ...;    // Signing key
EncryptionMaterialsProvider provider = new SymmetricStaticProvider(cek, macKey);

// To decrypt
SecretKey cek = ...;        // Encryption key
SecretKey macKey =  ...;    // Verification key
EncryptionMaterialsProvider provider = new SymmetricStaticProvider(cek, macKey);
```

Python

```
# You can provide encryption materials, decryption materials, or both
encrypt_keys = EncryptionMaterials(
    encryption_key = ...,
    signing_key = ...
)

decrypt_keys = DecryptionMaterials(
    decryption_key = ...,
    verification_key = ...
)

static_cmp = StaticCryptographicMaterialsProvider(
    encryption_materials=encrypt_keys
    decryption_materials=decrypt_keys
)
```

How it works

The Static Provider passes the encryption and signing keys that you supply to the item encryptor, where they are used directly to encrypt and sign your table items. Unless you supply different keys for each item, the same keys are used for every item.



Get encryption materials

This section describes in detail the inputs, outputs, and processing of the Static Materials Provider (Static CMP) when it receives a request for encryption materials.

Input (from the application)

- An encryption key – This must be a symmetric key, such as an Advanced Encryption Standard (AES) key.
- A signing key – This can be a symmetric key or an asymmetric key pair.

Input (from the item encryptor)

- DynamoDB encryption context (p. 10)

Output (to the item encryptor)

- The encryption key passed as input.
- The signing key passed as input.
- Actual material description: The requested material description (p. 9), if any, unchanged.

Get decryption materials

This section describes in detail the inputs, outputs, and processing of the Static Materials Provider (Static CMP) when it receives a request for decryption materials.

Although it includes separate methods for getting encryption materials and getting decryption materials, the behavior is the same.

Input (from the application)

- An encryption key – This must be a symmetric key, such as an Advanced Encryption Standard (AES) key.
- A signing key – This can be a symmetric key or an asymmetric key pair.

Input (from the item encryptor)

- DynamoDB encryption context (p. 10) (not used)

Output (to the item encryptor)

- The encryption key passed as input.
- The signing key passed as input.

Amazon DynamoDB Encryption Client available programming languages

The Amazon DynamoDB Encryption Client is available for the following programming languages. The language-specific libraries vary, but the resulting implementations are interoperable. For example, you can encrypt (and sign) an item with the Java client and decrypt the item with the Python client.

For more information, see the corresponding topic.

Topics

- Amazon DynamoDB Encryption Client for Java (p. 32)
- DynamoDB Encryption Client for Python (p. 41)

Amazon DynamoDB Encryption Client for Java

This topic explains how to install and use the Amazon DynamoDB Encryption Client for Java. For details about programming with the DynamoDB Encryption Client, see the Java examples (p. 37), the examples in the aws-dynamodb-encryption-java repository on GitHub, and the Javadoc for the DynamoDB Encryption Client.

> **Note**
> DynamoDB Encryption Client for Java versions 1.*x.x* are in end-of-support phase (p. 2) effective July 2022. Upgrade to a newer version as soon as possible.

Topics

- Prerequisites (p. 32)
- Installation (p. 33)
- Using the DynamoDB Encryption Client for Java (p. 33)
- Example code for the DynamoDB Encryption Client for Java (p. 37)

Prerequisites

Before you install the Amazon DynamoDB Encryption Client for Java, be sure you have the following prerequisites.

A Java development environment

You will need Java 8 or later. On the Oracle website, go to Java SE Downloads, and then download and install the Java SE Development Kit (JDK).

If you use the Oracle JDK, you must also download and install the Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files.

AWS SDK for Java

The DynamoDB Encryption Client requires the DynamoDB module of the AWS SDK for Java even if your application doesn't interact with DynamoDB. You can install the entire SDK or just this module. If you are using Maven, add `aws-java-sdk-dynamodb` to your `pom.xml` file.

For more information about installing and configuring the AWS SDK for Java, see AWS SDK for Java.

Installation

You can install the Amazon DynamoDB Encryption Client for Java in the following ways.

Manually

To install the Amazon DynamoDB Encryption Client for Java, clone or download the aws-dynamodb-encryption-java GitHub repository.

Using Apache Maven

The Amazon DynamoDB Encryption Client for Java is available through Apache Maven with the following dependency definition.

```
<dependency>
  <groupId>com.amazonaws</groupId>
  <artifactId>aws-dynamodb-encryption-java</artifactId>
  <version>version-number</version>
</dependency>
```

After you install the SDK, get started by looking at the example code in this guide and the DynamoDB Encryption Client Javadoc on GitHub.

Using the DynamoDB Encryption Client for Java

This topic explains some of the features of the DynamoDB Encryption Client in Java that might not be found in other programming language implementations.

For details about programming with the DynamoDB Encryption Client, see the Java examples (p. 37), the examples in the `aws-dynamodb-encryption-java repository` on GitHub, and the Javadoc for the DynamoDB Encryption Client.

Topics

- Item encryptors: AttributeEncryptor and DynamoDBEncryptor (p. 33)
- Configuring save behavior (p. 34)
- Attribute actions in Java (p. 34)
- Overriding table names (p. 36)

Item encryptors: AttributeEncryptor and DynamoDBEncryptor

The DynamoDB Encryption Client in Java has two item encryptors (p. 9): the lower-level DynamoDBEncryptor and the AttributeEncryptor (p. 33).

The `AttributeEncryptor` is a helper class that helps you use the DynamoDBMapper in the AWS SDK for Java with the `DynamoDB Encryptor` in the DynamoDB Encryption Client. When you use the

`AttributeEncryptor` with the `DynamoDBMapper`, it transparently encrypts and signs your items when you save them. It also transparently verifies and decrypts your items when you load them.

Configuring save behavior

You can use the `AttributeEncryptor` and `DynamoDBMapper` to add or edit table items with attributes that are signed only or encrypted and signed. For these tasks, we recommend that you configure it to use the `PUT` save behavior, as shown in the following example. Otherwise, you might not be able to decrypt your data.

```
DynamoDBMapperConfig mapperConfig =
 DynamoDBMapperConfig.builder().withSaveBehavior(SaveBehavior.PUT).build();
DynamoDBMapper mapper = new DynamoDBMapper(ddb, mapperConfig, new
 AttributeEncryptor(encryptor));
```

If you use the default save behavior, which updates the attributes in the table item, only attributes that have changed are included in the signature. That signature might not match the signature of the entire item that is calculated on decrypt.

You can also use the `CLOBBER` save behavior. This behavior is identical to the `PUT` save behavior except that it disables optimistic locking and overwrites the item in the table.

To see this code used in an example, see Using the DynamoDBMapper (p. 39) and the AwsKmsEncryptedObject.java example in the `aws-dynamodb-encryption-java` repository in GitHub.

Attribute actions in Java

Attribute actions (p. 9) determine which attribute values are encrypted and signed, which are only signed, and which are ignored. The method you use to specify attribute actions depends on whether you use the `DynamoDBMapper` and `AttributeEncryptor`, or the lower-level DynamoDBEncryptor.

> **Important**
> After you use your attribute actions to encrypt your table items, adding or removing attributes from your data model might cause a signature validation error that prevents you from decrypting your data. For a detailed explanation, see Changing your data model (p. 50).

Attribute actions for the DynamoDBMapper

When you use the `DynamoDBMapper` and `AttributeEncryptor`, you use annotations to specify the attribute actions. The DynamoDB Encryption Client uses the standard DynamoDB attribute annotations that define the attribute type to determine how to protect an attribute. By default, all attributes are encrypted and signed except for primary keys, which are signed but not encrypted.

> **Note**
> Do not encrypt the value of attributes with the @DynamoDBVersionAttribute annotation, although you can (and should) sign them. Otherwise, conditions that use its value will have unintended effects.

```
// Attributes are encrypted and signed
@DynamoDBAttribute(attributeName="Description")

// Partition keys are signed but not encrypted
@DynamoDBHashKey(attributeName="Title")

// Sort keys are signed but not encrypted
@DynamoDBRangeKey(attributeName="Author")
```

To specify exceptions, use the encryption annotations defined in the DynamoDB Encryption Client for Java. If you specify them at the class level, they become the default value for the class.

```
// Sign only
@DoNotEncrypt

// Do nothing; not encrypted or signed
@DoNotTouch
```

For example, these annotations sign but do not encrypt the `PublicationYear` attribute, and do not encrypt or sign the `ISBN` attribute value.

```
// Sign only (override the default)
@DoNotEncrypt
@DynamoDBAttribute(attributeName="PublicationYear")

// Do nothing (override the default)
@DoNotTouch
@DynamoDBAttribute(attributeName="ISBN")
```

Attribute actions for the DynamoDBEncryptor

To specify attribute actions when you use the DynamoDBEncryptor directly, create a `HashMap` object in which the name-value pairs represent attribute names and the specified actions.

The valid values are for the attribute actions are defined in the `EncryptionFlags` enumerated type. You can use `ENCRYPT` and `SIGN` together, use `SIGN` alone, or omit both. However, if you use `ENCRYPT` alone, the DynamoDB Encryption Client throws an error. You cannot encrypt an attribute that you don't sign.

```
ENCRYPT
SIGN
```

Warning
Do not encrypt the primary key attributes. They must remain in plaintext so DynamoDB can find the item without running a full table scan.

If you specify a primary key in the encryption context and then specify `ENCRYPT` in the attribute action for either primary key attribute, the DynamoDB Encryption Client throws an exception.

For example, the following Java code creates an `actions` HashMap that encrypts and signs all attributes in the `record` item. The exceptions are the partition key and sort key attributes, which are signed but not encrypted, and the `test` attribute, which is not signed or encrypted.

```
final EnumSet<EncryptionFlags> signOnly = EnumSet.of(EncryptionFlags.SIGN);
final EnumSet<EncryptionFlags> encryptAndSign = EnumSet.of(EncryptionFlags.ENCRYPT,
 EncryptionFlags.SIGN);
final Map<String, Set<EncryptionFlags>> actions = new HashMap<>();

for (final String attributeName : record.keySet()) {
  switch (attributeName) {
    case partitionKeyName: // no break; falls through to next case
    case sortKeyName:
      // Partition and sort keys must not be encrypted, but should be signed
      actions.put(attributeName, signOnly);
      break;
    case "test":
      // Don't encrypt or sign
      break;
    default:
      // Encrypt and sign everything else
      actions.put(attributeName, encryptAndSign);
      break;
```

```
  }
}
```

Then, when you call the encryptRecord method of the `DynamoDBEncryptor`, specify the map as the value of the `attributeFlags` parameter. For example, this call to `encryptRecord` uses the `actions` map.

```
// Encrypt the plaintext record
final Map<String, AttributeValue> encrypted_record = encryptor.encryptRecord(record,
 actions, encryptionContext);
```

Overriding table names

In the DynamoDB Encryption Client, the name of the DynamoDB table is an element of the DynamoDB encryption context (p. 10) that is passed to the encryption and decryption methods. When you encrypt or sign table items, the DynamoDB encryption context, including the table name, is cryptographically bound to the ciphertext. If the DynamoDB encryption context that is passed to the decrypt method doesn't match the DynamoDB encryption context that was passed to the encrypt method, the decrypt operation fails.

Occasionally, the name of a table changes, such as when you back up a table or perform a point-in-time recovery. When you decrypt or verify the signature of these items, you must pass in the same DynamoDB encryption context that was used to encrypt and sign the items, including the original table name. The current table name is not needed.

When you use the `DynamoDBEncryptor`, you assemble the DynamoDB encryption context manually. However, if you are using the `DynamoDBMapper`, the `AttributeEncryptor` creates the DynamoDB encryption context for you, including the current table name. To tell the `AttributeEncryptor` to create an encryption context with a different table name, use the `EncryptionContextOverrideOperator`.

For example, the following code creates instances of the cryptographic materials provider (CMP) and the `DynamoDBEncryptor`. Then it calls the `setEncryptionContextOverrideOperator` method of the `DynamoDBEncryptor`. It uses the `overrideEncryptionContextTableName` operator, which overrides one table name. When it is configured this way, the `AttributeEncryptor` creates a DynamoDB encryption context that includes `newTableName` in place of `oldTableName`. For a complete example, see EncryptionContextOverridesWithDynamoDBMapper.java.

```
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, keyArn);
final DynamoDBEncryptor encryptor = DynamoDBEncryptor.getInstance(cmp);

encryptor.setEncryptionContextOverrideOperator(EncryptionContextOperators.overrideEncryptionContextTabl
                oldTableName, newTableName));
```

When you call the load method of the `DynamoDBMapper`, which decrypts and verifies the item, you specify the original table name.

```
mapper.load(itemClass, DynamoDBMapperConfig.builder()

 .withTableNameOverride(DynamoDBMapperConfig.TableNameOverride.withTableNameReplacement(oldTableName))
                .build());
```

You can also use the `overrideEncryptionContextTableNameUsingMap` operator, which overrides multiple table names.

The table name override operators are typically used when decrypting data and verifying signatures. However, you can use them to set the table name in the DynamoDB encryption context to a different value when encrypting and signing.

Do not use the table name override operators if you are using the `DynamoDBEncryptor`. Instead, create an encryption context with the original table name and submit it to the decryption method.

Example code for the DynamoDB Encryption Client for Java

The following examples show you how to use the DynamoDB Encryption Client for Java to protect DynamoDB table items in your application. You can find more examples (and contribute your own) in the examples directory of the aws-dynamodb-encryption-java repository on GitHub.

Topics

- Using the DynamoDBEncryptor (p. 37)
- Using the DynamoDBMapper (p. 39)

Using the DynamoDBEncryptor

This example shows how to use the lower-level DynamoDBEncryptor with the Direct KMS Provider (p. 13). The Direct KMS Provider generates and protects its cryptographic materials under an AWS KMS key in AWS Key Management Service (AWS KMS) that you specify.

You can use any compatible cryptographic materials provider (p. 8) (CMP) with the `DynamoDBEncryptor`, and you can use the Direct KMS Provider with the `DynamoDBMapper` and AttributeEncryptor (p. 33).

See the complete code sample: AwsKmsEncryptedItem.java

Step 1: Create the Direct KMS Provider

Create an instance of the AWS KMS client with the specified region. Then, use the client instance to create an instance of the Direct KMS Provider with your preferred AWS KMS key.

This example uses the Amazon Resource Name (ARN) to identify the AWS KMS key, but you can use any valid key identifier.

```
final String keyArn = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
final String region = 'us-west-2'

final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, keyArn);
```

Step 2: Create an item

This example defines a `record` HashMap that represents a sample table item.

```
final String partitionKeyName = "partition_attribute";
final String sortKeyName = "sort_attribute";

final Map<String, AttributeValue> record = new HashMap<>();
record.put(partitionKeyName, new AttributeValue().withS("value1"));
record.put(sortKeyName, new AttributeValue().withN("55"));
record.put("example", new AttributeValue().withS("data"));
record.put("numbers", new AttributeValue().withN("99"));
record.put("binary", new AttributeValue().withB(ByteBuffer.wrap(new byte[]{0x00, 0x01,
 0x02})));
record.put("test", new AttributeValue().withS("test-value"));
```

Step 3: Create a DynamoDBEncryptor

Create an instance of the `DynamoDBEncryptor` with the Direct KMS Provider.

```
final DynamoDBEncryptor encryptor = DynamoDBEncryptor.getInstance(cmp);
```

Step 4: Create a DynamoDB encryption context

The DynamoDB encryption context (p. 10) contains information about the table structure and how it is encrypted and signed. If you use the `DynamoDBMapper`, the `AttributeEncryptor` creates the encryption context for you.

```
final String tableName = "testTable";

final EncryptionContext encryptionContext = new EncryptionContext.Builder()
    .withTableName(tableName)
    .withHashKeyName(partitionKeyName)
    .withRangeKeyName(sortKeyName)
    .build();
```

Step 5: Create the attribute actions object

Attribute actions (p. 9) determine which attributes of the item are encrypted and signed, which are only signed, and which are not encrypted or signed.

In Java, to specify attribute actions, you create a HashMap of attribute name and `EncryptionFlags` value pairs.

For example, the following Java code creates an `actions` HashMap that encrypts and signs all attributes in the `record` item, except for the partition key and sort key attributes, which are signed, but not encrypted, and the `test` attribute, which is not signed or encrypted.

```
final EnumSet<EncryptionFlags> signOnly = EnumSet.of(EncryptionFlags.SIGN);
final EnumSet<EncryptionFlags> encryptAndSign = EnumSet.of(EncryptionFlags.ENCRYPT,
 EncryptionFlags.SIGN);
final Map<String, Set<EncryptionFlags>> actions = new HashMap<>();

for (final String attributeName : record.keySet()) {
  switch (attributeName) {
    case partitionKeyName: // fall through to the next case
    case sortKeyName:
      // Partition and sort keys must not be encrypted, but should be signed
      actions.put(attributeName, signOnly);
      break;
    case "test":
      // Neither encrypted nor signed
      break;
    default:
      // Encrypt and sign all other attributes
      actions.put(attributeName, encryptAndSign);
      break;
  }
}
```

Step 6: Encrypt and sign the item

To encrypt and sign the table item, call the `encryptRecord` method on the instance of the `DynamoDBEncryptor`. Specify the table item (`record`), the attribute actions (`actions`), and the encryption context (`encryptionContext`).

```
final Map<String, AttributeValue> encrypted_record = encryptor.encryptRecord(record,
 actions, encryptionContext);
```

Step 7: Put the item in the DynamoDB table

Finally, put the encrypted and signed item in the DynamoDB table.

```
final AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();
ddb.putItem(tableName, encrypted_record);
```

Using the DynamoDBMapper

The following example shows you how to use the DynamoDB mapper helper class with the Direct KMS Provider (p. 13). The Direct KMS Provider generates and protects its cryptographic materials under an AWS KMS key in AWS Key Management Service (AWS KMS) that you specify.

You can use any compatible cryptographic materials provider (p. 8) (CMP) with the `DynamoDBMapper`, and you can use the Direct KMS Provider with the lower-level `DynamoDBEncryptor`.

See the complete code sample: AwsKmsEncryptedObject.java

Step 1: Create the Direct KMS Provider

Create an instance of the AWS KMS client with the specified region. Then, use the client instance to create an instance of the Direct KMS Provider with your preferred AWS KMS key.

This example uses the Amazon Resource Name (ARN) to identify the AWS KMS key, but you can use any valid key identifier.

```
final String keyArn = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
final String region = 'us-west-2'

final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, keyArn);
```

Step 2: Create the DynamoDB Encryptor and DynamoDBMapper

Use the Direct KMS Provider that you created in the previous step to create an instance of the DynamoDB Encryptor (p. 33). You need to instantiate the lower-level DynamoDB Encryptor to use the DynamoDB Mapper.

Next, create an instance of your DynamoDB database and a mapper configuration, and use them to create an instance of the DynamoDB Mapper.

Important
When using the `DynamoDBMapper` to add or edit signed (or encrypted and signed) items, configure it to use a save behavior (p. 34), such as `PUT`, that includes all attributes, as shown in the following example. Otherwise, you might not be able to decrypt your data.

```
final DynamoDBEncryptor encryptor = DynamoDBEncryptor.getInstance(cmp)
final AmazonDynamoDB ddb =
 AmazonDynamoDBClientBuilder.standard().withRegion(region).build();

DynamoDBMapperConfig mapperConfig =
 DynamoDBMapperConfig.builder().withSaveBehavior(SaveBehavior.PUT).build();
DynamoDBMapper mapper = new DynamoDBMapper(ddb, mapperConfig, new
 AttributeEncryptor(encryptor));
```

Step 3: Define your DynamoDB table

Next, define your DynamoDB table. Use annotations to specify the attribute actions (p. 34). This example creates a DynamoDB table, `ExampleTable`, and a `DataPoJo` class that represents table items.

In this sample table, the primary key attributes will be signed but not encrypted. This applies to the `partition_attribute`, which is annotated with `@DynamoDBHashKey`, and the `sort_attribute`, which is annotated with `@DynamoDBRangeKey`.

Attributes that are annotated with `@DynamoDBAttribute`, such as `some numbers`, will be encrypted and signed. The exceptions are attributes that use the `@DoNotEncrypt` (sign only) or `@DoNotTouch` (do not encrypt or sign) encryption annotations defined by the DynamoDB Encryption Client. For example, because the `leave me` attribute has a `@DoNotTouch` annotation, it will not be encrypted or signed.

```
@DynamoDBTable(tableName = "ExampleTable")
public static final class DataPoJo {
  private String partitionAttribute;
  private int sortAttribute;
  private String example;
  private long someNumbers;
  private byte[] someBinary;
  private String leaveMe;

  @DynamoDBHashKey(attributeName = "partition_attribute")
  public String getPartitionAttribute() {
    return partitionAttribute;
  }

  public void setPartitionAttribute(String partitionAttribute) {
    this.partitionAttribute = partitionAttribute;
  }

  @DynamoDBRangeKey(attributeName = "sort_attribute")
  public int getSortAttribute() {
    return sortAttribute;
  }

  public void setSortAttribute(int sortAttribute) {
    this.sortAttribute = sortAttribute;
  }

  @DynamoDBAttribute(attributeName = "example")
  public String getExample() {
    return example;
  }

  public void setExample(String example) {
    this.example = example;
  }

  @DynamoDBAttribute(attributeName = "some numbers")
  public long getSomeNumbers() {
    return someNumbers;
  }

  public void setSomeNumbers(long someNumbers) {
    this.someNumbers = someNumbers;
  }

  @DynamoDBAttribute(attributeName = "and some binary")
  public byte[] getSomeBinary() {
    return someBinary;
```

```
  }

  public void setSomeBinary(byte[] someBinary) {
    this.someBinary = someBinary;
  }

  @DynamoDBAttribute(attributeName = "leave me")
  @DoNotTouch
  public String getLeaveMe() {
    return leaveMe;
  }

  public void setLeaveMe(String leaveMe) {
    this.leaveMe = leaveMe;
  }

  @Override
  public String toString() {
    return "DataPoJo [partitionAttribute=" + partitionAttribute + ", sortAttribute="
        + sortAttribute + ", example=" + example + ", someNumbers=" + someNumbers
        + ", someBinary=" + Arrays.toString(someBinary) + ", leaveMe=" + leaveMe + "]";
  }
}
```

Step 4: Encrypt and save a table item

Now, when you create a table item and use the DynamoDB Mapper to save it, the item is automatically encrypted and signed before it is added to the table.

This example defines a table item called `record`. Before it is saved in the table, its attributes are encrypted and signed based on the annotations in the `DataPoJo` class. In this case, all attributes except for `PartitionAttribute`, `SortAttribute`, and `LeaveMe` are encrypted and signed. `PartitionAttribute` and `SortAttributes` are only signed. The `LeaveMe` attribute is not encrypted or signed.

To encrypt and sign the `record` item, and then add it to the `ExampleTable`, call the `save` method of the `DynamoDBMapper` class. Because your DynamoDB Mapper is configured to use the `PUT` save behavior, the item replaces any item with the same primary keys, instead of updating it. This ensures that the signatures match and you can decrypt the item when you get it from the table.

```
DataPoJo record = new DataPoJo();
record.setPartitionAttribute("is this");
record.setSortAttribute(55);
record.setExample("data");
record.setSomeNumbers(99);
record.setSomeBinary(new byte[]{0x00, 0x01, 0x02});
record.setLeaveMe("alone");

mapper.save(record);
```

DynamoDB Encryption Client for Python

This topic explains how to install and use the DynamoDB Encryption Client for Python. You can find the code in the aws-dynamodb-encryption-python repository on GitHub, including complete and tested sample code to help you get started.

Note
DynamoDB Encryption Client for Python versions 1.*x.x* and 2.*x.x* are in end-of-support phase (p. 2) effective July 2022. Upgrade to a newer version as soon as possible.

Topics

- Prerequisites (p. 42)
- Installation (p. 42)
- Using the DynamoDB Encryption Client for Python (p. 42)
- Example code for the DynamoDB Encryption Client for Python (p. 44)

Prerequisites

Before you install the Amazon DynamoDB Encryption Client for Python, be sure you have the following prerequisites.

A supported version of Python

Python 3.6 or later is required by the Amazon DynamoDB Encryption Client for Python versions 3.1.0 and later. To download Python, see Python downloads.

Earlier versions of the Amazon DynamoDB Encryption Client for Python support Python 2.7 and Python 3.4 and later, but we recommend that you use the latest version of the DynamoDB Encryption Client.

The pip installation tool for Python

Python 3.6 and later include **pip**, although you might want to upgrade it. For more information about upgrading or installing pip, see Installation in the **pip** documentation.

Installation

Use **pip** to install the Amazon DynamoDB Encryption Client for Python, as shown in the following examples.

To install the latest version

```
pip install dynamodb-encryption-sdk
```

For more details about using **pip** to install and upgrade packages, see Installing Packages.

The DynamoDB Encryption Client requires the cryptography library on all platforms. All versions of **pip** install and build the **cryptography** library on Windows. **pip** 8.1 and later installs and builds **cryptography** on Linux. If you are using an earlier version of **pip** and your Linux environment doesn't have the tools needed to build the **cryptography** library, you need to install them. For more information, see Building cryptography on Linux.

You can get the latest development version of the DynamoDB Encryption Client from the aws-dynamodb-encryption-python repository on GitHub.

After you install the DynamoDB Encryption Client, get started by looking at the example Python code in this guide.

Using the DynamoDB Encryption Client for Python

This topic explains some of the features of the DynamoDB Encryption Client for Python that might not be found in other programming language implementations. These features are designed to make it easier to use the DynamoDB Encryption Client in the most secure way. Unless you have an unusual use case, we recommend that you use them.

For details about programming with the DynamoDB Encryption Client, see the Python examples (p. 44) in this guide, the examples in the aws-dynamodb-encryption-python repository on GitHub, and the Python documentation for the DynamoDB Encryption Client.

Topics

- Client helper classes (p. 43)
- TableInfo class (p. 43)
- Attribute actions in Python (p. 44)

Client helper classes

The DynamoDB Encryption Client for Python includes several client helper classes that mirror the Boto 3 classes for DynamoDB. These helper classes are designed to make it easier to add encryption and signing to your existing DynamoDB application and avoid the most common problems, as follows:

- Prevent you from encrypting the primary key in your item, either by adding an override action for the primary key to the AttributeActions (p. 44) object, or by throwing an exception if your `AttributeActions` object explicitly tells the client to encrypt the primary key. If the default action in your `AttributeActions` object is `DO_NOTHING`, the client helper classes use that action for the primary key. Otherwise, they use `SIGN_ONLY`.
- Create a TableInfo object (p. 43) and populate the DynamoDB encryption context (p. 10) based on a call to DynamoDB. This helps to ensure that your DynamoDB encryption context is accurate and the client can identify the primary key.
- Support methods, such as `put_item` and `get_item`, that transparently encrypt and decrypt your table items when you write to or read from a DynamoDB table. Only the `update_item` method is unsupported.

You can use the client helper classes instead of interacting directly with the lower-level item encryptor (p. 9). Use these classes unless you need to set advanced options in the item encryptor.

The client helper classes include:

- EncryptedTable for applications that use the Table resource in DynamoDB to process one table at a time.
- EncryptedResource for applications that use the Service Resource class in DynamoDB for batch processing.
- EncryptedClient for applications that use the lower-level client in DynamoDB.

To use the client helper classes, the caller must have permission to call the DynamoDB DescribeTable operation on the target table.

TableInfo class

The TableInfo class is a helper class that represents a DynamoDB table, complete with fields for its primary key and secondary indexes. It helps you to get accurate, real-time information about the table.

If you use a client helper class (p. 43), it creates and uses a `TableInfo` object for you. Otherwise, you can create one explicitly. For an example, see Use the item encryptor (p. 46).

When you call the `refresh_indexed_attributes` method on a `TableInfo` object, it populates the property values of the object by calling the DynamoDB DescribeTable operation. Querying the table is much more reliable than hard-coding index names. The `TableInfo` class also includes an `encryption_context_values` property that provides the required values for the DynamoDB encryption context (p. 10).

To use the `refresh_indexed_attributes` method, the caller must have permission to call the DynamoDB DescribeTable operation on the target table.

Attribute actions in Python

Attribute actions (p. 9) tell the item encryptor which actions to perform on each attribute of the item. To specify attribute actions in Python, create an `AttributeActions` object with a default action and any exceptions for particular attributes. The valid values are defined in the `CryptoAction` enumerated type.

Important
After you use your attribute actions to encrypt your table items, adding or removing attributes from your data model might cause a signature validation error that prevents you from decrypting your data. For a detailed explanation, see Changing your data model (p. 50).

```
DO_NOTHING = 0
SIGN_ONLY = 1
ENCRYPT_AND_SIGN = 2
```

For example, this `AttributeActions` object establishes `ENCRYPT_AND_SIGN` as the default for all attributes, and specifies exceptions for the `ISBN` and `PublicationYear` attributes.

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
    attribute_actions={
        'ISBN': CryptoAction.DO_NOTHING,
        'PublicationYear': CryptoAction.SIGN_ONLY
    }
)
```

If you use a client helper class (p. 43), you don't need to specify an attribute action for the primary key attributes. The client helper classes prevent you from encrypting your primary key.

If you do not use a client helper class and the default action is `ENCRYPT_AND_SIGN`, you must specify an action for the primary key. The recommended action for primary keys is `SIGN_ONLY`. To make this easy, use the `set_index_keys` method, which uses SIGN_ONLY for primary keys, or DO_NOTHING, when that is the default action.

Warning
Do not encrypt the primary key attributes. They must remain in plaintext so DynamoDB can find the item without running a full table scan.

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
)
actions.set_index_keys(*table_info.protected_index_keys())
```

Example code for the DynamoDB Encryption Client for Python

The following examples show you how to use the DynamoDB Encryption Client for Python to protect DynamoDB data in your application. You can find more examples (and contribute your own) in the examples directory of the aws-dynamodb-encryption-python repository on GitHub.

Topics

- Use the EncryptedTable client helper class (p. 45)

- Use the item encryptor (p. 46)

Use the EncryptedTable client helper class

The following example shows you how to use the Direct KMS Provider (p. 13) with the `EncryptedTable` client helper class (p. 43). This example uses the same cryptographic materials provider (p. 8) as the Use the item encryptor (p. 46) example that follows. However, it uses the `EncryptedTable` class instead of interacting directly with the lower-level item encryptor (p. 9).

By comparing these examples, you can see the work that the client helper class does for you. This includes creating the DynamoDB encryption context (p. 10) and making sure the primary key attributes are always signed, but never encrypted. To create the encryption context and discover the primary key, the client helper classes call the DynamoDB DescribeTable operation. To run this code, you must have permission to call this operation.

See the complete code sample: aws_kms_encrypted_table.py

Step 1: Create the table

Start by creating an instance of a standard DynamoDB table with the table name.

```
table_name='test-table'
table = boto3.resource('dynamodb').Table(table_name)
```

Step 2: Create a cryptographic materials provider

Create an instance of the cryptographic materials provider (p. 12) (CMP) that you selected.

This example uses the Direct KMS Provider (p. 13), but you can use any compatible CMP. To create a Direct KMS Provider, specify an AWS KMS key. This example uses the Amazon Resource Name (ARN) of the AWS KMS key, but you can use any valid key identifier.

```
kms_key_id='arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=kms_key_id)
```

Step 3: Create the attribute actions object

Attribute actions (p. 9) tell the item encryptor which actions to perform on each attribute of the item. The `AttributeActions` object in this example encrypts and signs all items except for the `test` attribute, which is ignored.

Do not specify attribute actions for the primary key attributes when you use a client helper class. The `EncryptedTable` class signs, but never encrypts, the primary key attributes.

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
    attribute_actions={'test': CryptoAction.DO_NOTHING}
)
```

Step 4: Create the encrypted table

Create the encrypted table using the standard table, the Direct KMS Provider, and the attribute actions. This step completes the configuration.

```
encrypted_table = EncryptedTable(
```

```
    table=table,
    materials_provider=kms_cmp,
    attribute_actions=actions
)
```

Step 5: Put the plaintext item in the table

When you call the `put_item` method on the `encrypted_table`, your table items are transparently encrypted, signed, and added to your DynamoDB table.

First, define the table item.

```
plaintext_item = {
    'partition_attribute': 'value1',
    'sort_attribute': 55
    'example': 'data',
    'numbers': 99,
    'binary': Binary(b'\x00\x01\x02'),
    'test': 'test-value'
}
```

Then, put it in the table.

```
encrypted_table.put_item(Item=plaintext_item)
```

To get the item from the DynamoDB table in its encrypted form, call the `get_item` method on the `table` object. To get the decrypted item, call the `get_item` method on the `encrypted_table` object.

Use the item encryptor

This example shows you how to interact directly with the item encryptor (p. 9) in the DynamoDB Encryption Client when encrypting table items, instead of using the client helper classes (p. 43) that interact with the item encryptor for you.

When you use this technique, you create the DynamoDB encryption context and configuration object (`CryptoConfig`) manually. Also, you encrypt the items in one call and put them in your DynamoDB table in a separate call. This allows you to customize your `put_item` calls and use the DynamoDB Encryption Client to encrypt and sign structured data that is never sent to DynamoDB.

This example uses the Direct KMS Provider (p. 13), but you can use any compatible CMP.

See the complete code sample: aws_kms_encrypted_item.py

Step 1: Create the table

Start by creating an instance of a standard DynamoDB table resource with the table name.

```
table_name='test-table'
table = boto3.resource('dynamodb').Table(table_name)
```

Step 2: Create a cryptographic materials provider

Create an instance of the cryptographic materials provider (p. 12) (CMP) that you selected.

This example uses the Direct KMS Provider (p. 13), but you can use any compatible CMP. To create a Direct KMS Provider, specify an AWS KMS key. This example uses the Amazon Resource Name (ARN) of the AWS KMS key, but you can use any valid key identifier.

```
kms_key_id='arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=kms_key_id)
```

Step 3: Use the TableInfo helper class

To get information about the table from DynamoDB, create an instance of the TableInfo (p. 43) helper class. When you work directly with the item encryptor, you need to create a `TableInfo` instance and call its methods. The client helper classes (p. 43) do this for you.

The `refresh_indexed_attributes` method of `TableInfo` uses the DescribeTable DynamoDB operation to get real-time, accurate information about the table. This includes its primary key and its local and global secondary indexes. The caller needs to have permission to call `DescribeTable`.

```
table_info = TableInfo(name=table_name)
table_info.refresh_indexed_attributes(table.meta.client)
```

Step 4: Create the DynamoDB encryption context

The DynamoDB encryption context (p. 10) contains information about the table structure and how it is encrypted and signed. This example creates a DynamoDB encryption context explicitly, because it interacts with the item encryptor. The client helper classes (p. 43) create the DynamoDB encryption context for you.

To get the partition key and sort key, you can use the properties of the TableInfo (p. 43) helper class.

```
index_key = {
    'partition_attribute': 'value1',
    'sort_attribute': 55
}

encryption_context = EncryptionContext(
    table_name=table_name,
    partition_key_name=table_info.primary_index.partition,
    sort_key_name=table_info.primary_index.sort,
    attributes=dict_to_ddb(index_key)
)
```

Step 5: Create the attribute actions object

Attribute actions (p. 9) tell the item encryptor which actions to perform on each attribute of the item. The `AttributeActions` object in this example encrypts and signs all items except for the primary key attributes, which are signed, but not encrypted, and the `test` attribute, which is ignored.

When you interact directly with the item encryptor and your default action is `ENCRYPT_AND_SIGN`, you must specify an alternative action for the primary key. You can use the `set_index_keys` method, which uses `SIGN_ONLY` for the primary key, or it uses `DO_NOTHING` if it's the default action.

To specify the primary key, this example uses the index keys in the TableInfo (p. 43) object, which is populated by a call to DynamoDB. This technique is safer than hard-coding primary key names.

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
    attribute_actions={'test': CryptoAction.DO_NOTHING}
)
```

```
actions.set_index_keys(*table_info.protected_index_keys())
```

Step 6: Create the configuration for the item

To configure the DynamoDB Encryption Client, use the objects that you just created in a CryptoConfig configuration for the table item. The client helper classes create the CryptoConfig for you.

```
crypto_config = CryptoConfig(
    materials_provider=kms_cmp,
    encryption_context=encryption_context,
    attribute_actions=actions
)
```

Step 7: Encrypt the item

This step encrypts and signs the item, but it doesn't put it in the DynamoDB table.

When you use a client helper class, your items are transparently encrypted and signed, and then added to your DynamoDB table when you call the `put_item` method of the helper class. When you use the item encryptor directly, the encrypt and put actions are independent.

First, create a plaintext item.

```
plaintext_item = {
    'partition_attribute': 'value1',
    'sort_key': 55,
    'example': 'data',
    'numbers': 99,
    'binary': Binary(b'\x00\x01\x02'),
    'test': 'test-value'
}
```

Then, encrypt and sign it. The `encrypt_python_item` method requires the `CryptoConfig` configuration object.

```
encrypted_item = encrypt_python_item(plaintext_item, crypto_config)
```

Step 8: Put the item in the table

This step puts the encrypted and signed item in the DynamoDB table.

```
table.put_item(Item=encrypted_item)
```

To view the encrypted item, call the `get_item` method on the original `table` object, instead of the `encrypted_table` object. It gets the item from the DynamoDB table without verifying and decrypting it.

```
encrypted_item = table.get_item(Key=partition_key)['Item']
```

The following image shows part of an example encrypted and signed table item.

The encrypted attribute values are binary data. The names and values of the primary key attributes (`partition_attribute` and `sort_attribute`) and the `test` attribute remain in plaintext. The output also shows the attribute that contains the signature (`*amzn-ddb-map-sig*`) and the materials description attribute (p. 9) (`*amzn-ddb-map-desc*`).

```
{
    '*amzn-ddb-map-desc*': Binary(b'\x00\x00\x00\x00\x00\x00\x10amzn-ddb-env-
\x00\x00\x00\xe0AQEBAHhA84wnXjEJdBbBBylRUFcZZK2j7xwh6UyLoL28nQ
+0FAAAAH4wfAYJKoZIhvcNAQcGoG8wbQIBADBoBgkqhkiG9w0BBwEwHgYJYIZIAWUDBAEuMBEEDPeFBydr
izYl0R0C4M7wAK6E1/N/bgTmHI=\x00\x00\x00\x17amzn-ddb-map-signingAlg\x00\x00\x00\nHm
\x00\x00\x00\x11/CBC/PKCS5Padding\x00\x00\x00\x10amzn-ddb-sig-alg\x00\x00\x00\x0eE
\x00\x00\x00\x0faws-kms-ec-attr\x00\x00\x00\x06*keys*'),
    '*amzn-ddb-map-sig*': Binary(b"\xd3\xc6\xc7\n\xb7#\x13\xd1Y\xea\xe4.|^\xbd\xdf
    'binary': Binary(b'!"\xc5\x92\xd7\x13\x1d\xe8Bs\x9b\x7f\xa8\x8e\x9c\xcf\x10\x
    'example': Binary(b"'b\x933\x9a+s\xf1\xd6a\xc5\xd5\x1aZ\xed\xd6\xce\xe9X\xf0T
    'numbers': Binary(b'\xd5\xa0\\d\xcc\x85\xf5\x1e\xb9-f!\xb9\xb8\x8a\x1aT\xbaq\x
    'partition_attribute': 'value1',
    'sort_attribute': 55,
    'test': 'test-value'
}
```

Changing your data model

Every time you encrypt or decrypt an item, you need to provide attribute actions (p. 9) that tell the DynamoDB Encryption Client which attributes to encrypt and sign, which attributes to sign (but not encrypt), and which to ignore. Attribute actions are not saved in the encrypted item and the DynamoDB Encryption Client does not update your attribute actions automatically.

> **Important**
> The DynamoDB Encryption Client does not support the encryption of existing, unencrypted DynamoDB table data.

Whenever you change your data model, that is, when you add or remove attributes from your table items, you risk an error. If the attribute actions that you specify do not account for all attributes in the item, the item might not be encrypted and signed the way that you intend. More importantly, if the attribute actions that you provide when decrypting an item differ from the attribute actions that you provided when encrypting the item, the signature verification might fail.

For example, if the attribute actions used to encrypt the item tell it to sign the `test` attribute, the signature in the item will include the `test` attribute. But if the attribute actions used to decrypt the item do not account for the `test` attribute, the verification will fail because the client will try to verify a signature that does not include the `test` attribute.

This is a particular problem when multiple applications read and write the same DynamoDB items because the DynamoDB Encryption Client must calculate the same signature for items in all applications. It's also a problem for any distributed application because changes in attribute actions must propagate to all hosts. Even if your DynamoDB tables are accessed by one host in one process, establishing a best practice process will help prevent errors if the project ever becomes more complex.

To avoid signature validation errors that prevent you from reading your table items, use the following guidance.

- Adding an attribute (p. 50) — If the new attribute changes your attribute actions, fully deploy the attribute action change before including the new attribute in an item.
- Removing an attribute (p. 52) — If you stop using an attribute in your items, do not change your attribute actions.
- Changing the action — After you have used an attribute actions configuration to encrypt your table items, you cannot safely change the default action or the action for an existing attribute without re-encrypting every item in your table.

Signature validation errors can be extremely difficult to resolve, so the best approach is to prevent them.

Topics

- Adding an attribute (p. 50)
- Removing an attribute (p. 52)

Adding an attribute

When you add a new attribute to table items, you might need to change your attribute actions. To prevent signature validation errors, we recommend that you implement this change in a two-stage process. Verify that the first stage is complete before starting the second stage.

1. Change the attribute actions in all applications that read or write to the table. Deploy these changes and confirm that the update has been propagated to all destination hosts.
2. Write values to the new attribute in your table items.

This two-stage approach ensures that all applications and hosts have the same attribute actions, and will calculate the same signature, before any encounter the new attribute. This is important even when the action for the attribute is *Do nothing* (don't encrypt or sign), because the default for some encryptors is to encrypt and sign.

The following examples show the code for the first stage in this process. They add a new item attribute, `link`, which stores a link to another table item. Because this link must remain in plain text, the example assigns it the sign-only action. After fully deploying this change and then verifying that all applications and hosts have the new attribute actions, you can begin to use the `link` attribute in your table items.

Java DynamoDB Mapper

When using the `DynamoDB Mapper` and `AttributeEncryptor`, by default, all attributes are encrypted and signed except for primary keys, which are signed but not encrypted. To specify a sign-only action, use the `@DoNotEncrypt` annotation.

This example uses the `@DoNotEncrypt` annotation for the new `link` attribute.

```
@DynamoDBTable(tableName = "ExampleTable")
public static final class DataPoJo {
  private String partitionAttribute;
  private int sortAttribute;
  private String link;

  @DynamoDBHashKey(attributeName = "partition_attribute")
  public String getPartitionAttribute() {
    return partitionAttribute;
  }

  public void setPartitionAttribute(String partitionAttribute) {
    this.partitionAttribute = partitionAttribute;
  }

  @DynamoDBRangeKey(attributeName = "sort_attribute")
  public int getSortAttribute() {
    return sortAttribute;
  }

  public void setSortAttribute(int sortAttribute) {
    this.sortAttribute = sortAttribute;
  }

  @DynamoDBAttribute(attributeName = "link")
  @DoNotEncrypt
  public String getLink() {
    return link;
  }

  public void setLink(String link) {
    this.link = link;
  }

  @Override
  public String toString() {
    return "DataPoJo [partitionAttribute=" + partitionAttribute + ",
        sortAttribute=" + sortAttribute + ",
        link=" + link + "]";
  }
```

```
}
```

Java DynamoDB encryptor

In the lower-level DynamoDB encryptor, you must set actions for each attribute. This example uses a switch statement where the default is `encryptAndSign` and exceptions are specified for the partition key, sort key, and the new `link` attribute. In this example, if the link attribute code was not fully deployed before it was used, the link attribute would be encrypted and signed by some applications, but only signed by others.

```
for (final String attributeName : record.keySet()) {
    switch (attributeName) {
        case partitionKeyName:
            // fall through to the next case
        case sortKeyName:
            // partition and sort keys must be signed, but not encrypted
            actions.put(attributeName, signOnly);
            break;
        case "link":
            // only signed
            actions.put(attributeName, signOnly);
            break;
        default:
            // Encrypt and sign all other attributes
            actions.put(attributeName, encryptAndSign);
            break;
    }
}
```

Python

In the DynamoDB Encryption Client for Python, you can specify a default action for all attributes and then specify exceptions.

If you use a Python client helper class (p. 43), you don't need to specify an attribute action for the primary key attributes. The client helper classes prevent you from encrypting your primary key. However, if you are not using a client helper class, you must set the SIGN_ONLY action on your partition key and sort key. If you accidentally encrypt your partition or sort key, you won't be able to recover your data without a full table scan.

This example specifies an exception for the new `link` attribute, which gets the `SIGN_ONLY` action.

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
    attribute_actions={
      'example': CryptoAction.DO_NOTHING,
      'link': CryptoAction.SIGN_ONLY
    }
)
```

Removing an attribute

If you no longer need an attribute in items that have been encrypted with the DynamoDB Encryption Client, you can stop using the attribute. However, do not delete or change the action for that attribute. If you do, and then encounter an item with that attribute, the signature calculated for the item will not match the original signature, and the signature validation will fail.

Although you might be tempted to remove all traces of the attribute from your code, add a comment that the item is no longer used instead of deleting it. Even if you do a full table scan to delete all

instances of the attribute, an encrypted item with that attribute might be cached or in process somewhere in your configuration.

Troubleshooting issues in your DynamoDB Encryption Client application

This section describes problems that you might encounter when using the DynamoDB Encryption Client and offers suggestions for resolving them.

To provide feedback on the DynamoDB Encryption Client, file an issue in the aws-dynamodb-encryption-java or aws-dynamodb-encryption-python GitHub repository.

To provide feedback on this documentation, use the feedback link on any page. You can also file an issue or contribute to aws-dynamodb-encryption-docs, the open-source repository for this documentation on GitHub.

Topics

- Access denied (p. 54)
- Signature verification fails (p. 55)
- Issues with older version global tables (p. 55)
- Poor performance of the Most Recent Provider (p. 56)

Access denied

Problem: Your application is denied access to a resource that it needs.

Suggestion: Learn about the required permissions and add them to the security context in which your application runs.

Details

To run an application that uses the a DynamoDB Encryption Client library, the caller must have permission to use its components. Otherwise, they will be denied access to the required elements.

- The DynamoDB Encryption Client does not require an Amazon Web Services (AWS) account or depend on any AWS service. However, if your application uses AWS, you need an AWS account and users who have permission to use the account.
- The DynamoDB Encryption Client does not require Amazon DynamoDB. However, If the application that uses the client creates DynamoDB tables, puts items into a table, or gets items from a table, the caller must have permission to use the required DynamoDB operations in your AWS account. For details, see the access control topics in the *Amazon DynamoDB Developer Guide*.
- If your application uses a client helper class (p. 43) in the DynamoDB Encryption Client for Python, the caller must have permission to call the DynamoDB DescribeTable operation.
- The DynamoDB Encryption Client does not require AWS Key Management Service(AWS KMS). However, if your application uses a Direct KMS Materials Provider (p. 13), or it uses a Most Recent Provider (p. 21)

with a provider store that uses AWS KMS, the caller must have permission to use the AWS KMS GenerateDataKey and Decrypt operations.

Signature verification fails

Problem: An item cannot be decrypted because signature verification fails. The item also might not be encrypted and signed as you intend.

Suggestion: Be sure that the attribute actions that you provide account for all attributes in the item. When decrypting an item, be sure to provide attribute actions that match the actions used to encrypt the item.

Details

The attribute actions (p. 9) that you provide tell the DynamoDB Encryption Client which attributes to encrypt and sign, which attributes to sign (but not encrypt), and which to ignore.

If the attribute actions that you specify do not account for all attributes in the item, the item might not be encrypted and signed the way that you intend. More importantly, if your attribute actions do not account for all attributes in the item, the item might not be encrypted and signed the way that you intend.

If the attribute actions that you provide when decrypting an item differ from the attribute actions that you provided when encrypting the item, the signature verification might fail. This is a particular problem for distributed applications in which new attribute actions might not have propagated to all hosts.

Signature validation errors are difficult to resolve. For help preventing them, take extra precautions when changing your data model. For details, see Changing your data model (p. 50).

Issues with older version global tables

Problem: Items in an older version Amazon DynamoDB global table cannot be decrypted because signature verification fails.

Suggestion: Set attribute actions so the reserved replication fields are not encrypted or signed.

Details

You can use the DynamoDB Encryption Client with DynamoDB global tables. We recommend that you use global tables with a multi-Region KMS key and replicate the KMS key into all AWS Regions where the global table is replicated.

Beginning with global tables version 2019.11.21, you can use global tables with the DynamoDB Encryption Client without any special configuration. However, if you use global tables version 2017.11.29, you must ensure that reserved replication fields are not encrypted or signed.

If you are using the global tables version 2017.11.29, you must set the attribute actions for the following attributes to `DO_NOTHING` in Java (p. 34) or `@DoNotTouch` in Python (p. 44).

- `aws:rep:deleting`
- `aws:rep:updatetime`
- `aws:rep:updateregion`

If you are using any other version of global tables, no action is required.

Poor performance of the Most Recent Provider

Problem: Your application is less responsive, especially after updating to a newer version of the DynamoDB Encryption Client.

Suggestion: Adjust the time-to-live value and cache size.

Details

The Most Recent Provider is designed to improve the performance of applications that use the DynamoDB Encryption Client by allowing limited reuse of cryptographic materials. When you configure the Most Recent Provider for your application, you have to balance improved performance with the security concerns that arise from caching and reuse.

In newer versions of the DynamoDB Encryption Client, the time-to-live (TTL) value determines how long cached cryptographic material providers (CMPs) can be used. The TTL also determines how often the Most Recent Provider checks for a new version of the CMP.

If your TTL is too long, your application might violate your business rules or security standards. If your TTL is too brief, frequent calls to the provider store can cause your provider store to throttle requests from your application and other applications that share your service account. To resolve this issue, adjust the TTL and cache size to a value that meets your latency and availability goals and conforms to your security standards. For details, see Setting a time-to-live value (p. 25).

Document history for the Amazon DynamoDB Encryption Client Developer Guide

The following table describes significant changes to this documentation. In addition to these major changes, we also update the documentation frequently to improve the descriptions and examples, and to address the feedback that you send to us. To be notified about significant changes, subscribe to the RSS feed.

update-history-change	update-history-description	update-history-date
Documentation change	Replace the AWS Key Management Service term *customer master key (CMK)* with *AWS KMS key* and *KMS key*.	August 30, 2021
New feature	Added support for AWS Key Management Service (AWS KMS) multi-Region keys. Multi-Region keys are AWS KMS keys in different AWS Regions that can be used interchangeably because they have the same key ID and key material.	June 8, 2021
New example	Added example of using the DynamoDBMapper in Java.	September 6, 2018
Python support	Added support for Python, in addition to Java.	May 2, 2018
Initial release	Initial release of this documentation.	May 2, 2018