Exercise 1: Add the AWS Encryption SDK

In this section, you will add client-side encryption to the Busy Engineer's Document Bucket using the AWS Encryption SDK and AWS KMS.

Background

In Getting Started [../getting-started/], you set up your Busy Engineer's Document Bucket environment and selected a workshop language.

Now you will add the AWS Encryption SDK to encrypt objects on the client, before they are transmitted off of the host machine to the internet. You will use AWS KMS to provide a `data key` for each object, using a CMK that you set up in Getting Started [../getting-started/].

Let's Go!

Starting Directory

Make sure you are in the `exercises` directory for the language of your choice:

Java Typescript Node.JS JavaScript Node.JS Python

```
1 cd ~/environment/workshop/exercises/java/add-esdk-start
```

Step 1: Add the ESDK Dependency

Look for `ADD-ESDK-START` comments in the code to help orient yourself.

Start by adding the Encryption SDK dependency to the code.

Java Typescript Node.JS JavaScript Node.JS Python

```java
// Edit ./src/main/java/sfw/example/esdkworkshop/Api.java
package sfw.example.esdkworkshop;

// ADD-ESDK-START: Add the ESDK Dependency
import com.amazonaws.encryptionsdk.AwsCrypto;
import com.amazonaws.encryptionsdk.CryptoResult;
import com.amazonaws.encryptionsdk.MasterKey;
import com.amazonaws.encryptionsdk.MasterKeyProvider;
import com.amazonaws.encryptionsdk.kms.KmsMasterKey;

...
private final String tableName;
private final String bucketName;
// ADD-ESDK-START: Add the ESDK Dependency
private final AwsCrypto awsEncryptionSdk;
private final MasterKeyProvider mkp;

...

public Api(
    AmazonDynamoDB ddbClient,
    String tableName,
    AmazonS3 s3Client,
    String bucketName,
    // ADD-ESDK-START: Add the ESDK Dependency
    MasterKeyProvider<? extends MasterKey> mkp) {
  this.ddbClient = ddbClient;
  this.tableName = tableName;
  this.s3Client = s3Client
  // ADD-ESDK-START: Add the ESDK Dependency
  this.awsEncryptionSdk = new AwsCrypto();
  this.mkp = mkp;
}

// Save and close.
// Edit ./src/main/java/sfw/example/esdkworkshop/Api.java
package sfw.example.esdkworkshop;

// ADD-ESDK-START: Add the ESDK Dependency
import com.amazonaws.encryptionsdk.kms.KmsMasterKeyProvider;
```

```
40  // Save and close.
41
```

What Happened?

1. You added a dependency on the AWS Encryption SDK library in your code
2. You changed the API to expect that a Keyring or Master Key Provider will be passed to your code to use in `store` and `retrieve` operations

Step 2: Add Encryption to `store`

Now that you have the AWS Encryption SDK imported, start encrypting your data before storing it.

Java Typescript Node.JS JavaScript Node.JS Python

```
1  // Edit ./src/main/java/sfw/example/esdkworkshop/Api.java
2  public PointerItem store(byte[] data, Map<String, String>
3  context) {
4      // ADD-ESDK-START: Add Encryption to store
5      CryptoResult<byte[], KmsMasterKey> encryptedMessage =
6  awsEncryptionSdk.encryptData(mkp, data);
7      DocumentBundle bundle =
8
   DocumentBundle.fromDataAndContext(encryptedMessage.getResult(),
   context);
       writeItem(bundle.getPointer());
       ...
```

What Happened?

The application will use the AWS Encryption SDK to encrypt your data client-side under a CMK before storing it by:

1. Requesting a new data key using your Keyring or Master Key Provider
2. Encrypting your data with the returned data key
3. Returning your encrypted data in the AWS Encryption SDK message format

4. Extracting the ciphertext from the AWS Encryption SDK message
5. Passing the ciphertext to the AWS S3 SDK for storage in S3

Step 3: Add Decryption to `retrieve`

Now that the application encypts your data before storing it, it will need to decrypt your data before returning it to the caller (at least for the data to be useful, anyway).

Java Typescript Node.JS JavaScript Node.JS Python

```
// Edit ./src/main/java/sfw/example/esdkworkshop/Api.java
// Find retrieve(...)
    byte[] data = getObjectData(key);
    // ADD-ESDK-START: Add Decryption to retrieve
    CryptoResult<byte[], KmsMasterKey> decryptedMessage =
awsEncryptionSdk.decryptData(mkp, data);
    ...
    return
DocumentBundle.fromDataAndPointer(decryptedMessage.getResult(),
pointer);
```

What Happened?

The application now decrypts data client-side, as well.

The data returned from S3 for `retrieve` is encrypted. Before returning that data to the user, you added a call to the AWS Encryption SDK to decrypt the data. Under the hood, the Encryption SDK is:

1. Reading the AWS Encryption SDK formatted encrypted message
2. Calling KMS to request to decrypt your message's encrypted data key using the Faythe CMK
3. Using the decrypted data key to decrypt the message
4. Returning the message plaintext and Encryption SDK headers to you

Step 4: Configure the Faythe CMK in the Encryption SDK

Now that you have declared your dependencies and updated your code to encrypt and decrypt data, the final step is to pass through the configuration to the AWS Encryption SDK to start using your KMS CMKs to protect your data.

Java Typescript Node.JS JavaScript Node.JS Python

```
// Edit ./src/main/java/sfw/example/esdkworkshop/Api.java
    AmazonS3 s3Client = AmazonS3ClientBuilder.defaultClient();

    // ADD-ESDK-START: Configure the Faythe CMK in the
Encryption SDK
    // Load configuration of KMS resources
    String faytheCMK = state.contents.FaytheCMK;

    // Set up the Master Key Provider to use KMS
    KmsMasterKeyProvider mkp =

KmsMasterKeyProvider.builder().withKeysForEncryption(faytheCMK).bu


    return new Api(ddbClient, tableName, s3Client, bucketName,
mkp);
```

What Happened?

In Getting Started [../getting-started/], you launched CloudFormation stacks for CMKs. One of these CMKs was nicknamed Faythe. As part of launching these templates, the CMK's Amazon Resource Name (ARN) was written to a configuration file on disk, the `state` variable that is loaded and parsed.

Now Faythe's ARN is pulled into a variable, and used to initialize a Keyring or Master Key Provider that will use the Faythe CMK. That new Keyring/Master Key Provider is passed into your API, and you are set to start encrypting and decrypting with KMS and the Encryption SDK.

Checking Your Work

Want to check your progress, or compare what you've done versus a finished example?

Check out the code in one of the `-complete` folders to compare.

Java Typescript Node.JS JavaScript Node.JS Python

```
cd ~/environment/workshop/exercises/java/add-esdk-complete
```

Try it Out

Now that the code is written, let's load it up and try it out.

If you'd like to try a finished example, use your language's `-complete` directory as described above.

Experiment using the API as much as you like.

To get started, here are some things to try:

- Compare CloudTrail Logs for usages of Faythe [https://us-east-2.console.aws.amazon.com/cloudtrail/home?region=us-east-2#] when you encrypt messages of different sizes (small, medium, large)
- Take a look at the contents of your S3 Document Bucket [https://s3.console.aws.amazon.com/s3/home] to inspect the raw object

For more things to try, check out Explore Further [#explore-further], below.

Java JavaScript Node.JS JavaScript Node.JS CLI Typescript Node.JS Typescript Node.JS CLI Python

```
// Compile your code
mvn compile

// To use the API programmatically, use this target to launch jshell
mvn jshell:run
/open startup.jsh
Api documentBucket = App.initializeDocumentBucket();
```

```
documentBucket.list();
documentBucket.store("Store me in the Document
Bucket!".getBytes());
for (PointerItem item : documentBucket.list()) {
    DocumentBundle document =
documentBucket.retrieve(item.partitionKey().getS());
    System.out.println(document.toString());
}
// Ctrl+D to exit jshell

// Or, to run logic that you write in App.java, use this target
after compile
mvn exec:java
```

Explore Further

- **Leveraging the Message Format** - The AWS Encryption SDK Message Format [https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/message-format.html] is an open standard. Can you write something to detect whether an entry in the Document Bucket has been encrypted in this format or not, and retrieve or decrypt appropriately?
- **More Test Content** - Small test strings are enough to get started, but you might be curious to see what the behavior and performance looks like with larger documents. What if you add support for loading files to and from disk to the Document Bucket?
- **Configuration Glue** - If you are curious how the Document Bucket is configured, take a peek at `~/environment/workshop/cdk/Makefile` and the `make state` target, as well as `config.toml` in the exercises root `~/environment/workshop/exercises/config.toml`. The Busy Engineer's Document Bucket uses a base TOML [https://github.com/toml-lang/toml] file to set standard names for all CloudFormation resources and a common place to discover the real deployed set. Then it uses the AWS Cloud Development Kit (CDK) to deploy the resources and write out their identifiers to the state file. Applications use the base TOML file `config.toml` to locate the state file and pull the expected resource names. And that's how the system bootstraps all the resources it needs!

Next exercise

Now that you are encrypting and decrypting, how about adding Multiple CMKs [../multi-cmk/]?