

Client-Side Field Level Encryption Guide

On this page

- Who Is This Guide For?
- Introduction
 - Scenario
 - Comparison of Security Features
- Implementation
 - Requirements
 - A. Create a Master Key
 - B. Create a Data Encryption Key
 - C. Specify Encrypted Fields Using JSON Schema
 - D. Create the MongoDB Client
 - E. Perform Encrypted Read/Write Operations
- Summary
- Additional Information
 - Download Example Project
 - Move to Production
 - Further Reading



Who Is This Guide For?

This use case guide is an introduction to implementing automatic Client-Side Field Level Encryption using supported MongoDB drivers and is intended for **full-stack developers**. The guide presents the following information in the context of a real-world scenario:

- **How Client-Side Field Level Encryption works** (Introduction)
- **Reasons to choose this security feature** (Comparison of Security Features)
- **How to implement Client-Side Field Level Encryption with**

the MongoDB driver (Implementation)

DOWNLOAD THE CODE:

For a runnable example of all the functionality demonstrated in this guide, see the Download Example Project section.

Introduction

Applications frequently use and store sensitive data such as confidential personal details, payment information, or proprietary data. In some jurisdictions, this type of data is subject to governance, privacy, and security compliance mandates. Unauthorized access of sensitive data or a failure to comply with a mandate often results in significant reputation damage and financial penalties. Therefore, it is important to keep sensitive data secure.

MongoDB offers several methods that protect your data from unauthorized access including:

- Role-based access control
- TLS/SSL network transport encryption
- Encryption at rest

Another MongoDB feature that prevents unauthorized access of data is Client-Side Field Level Encryption (CSFLE) [↗](#). This feature allows a developer to selectively encrypt individual fields of a document on the client-side before it is sent to the server. This keeps the encrypted data private from the providers hosting the database as well as any user that has direct access to the database.

This guide provides steps for setup and implementation of CSFLE with a practical example.

NOTE:

Automatic Client-Side Field Level Encryption is available starting in MongoDB 4.2 Enterprise only.

Scenario

In this scenario, we secure sensitive data on a Medical Care Management System which stores patients' personal information, insurance information, and medical records for a fictional company, *MedcoMD*. None of the patient data is public, and certain data such as their social security number (SSN, a US government-issued id number), insurance policy number, and vital sign measurements are particularly sensitive and subject to privacy compliance. It is important for the company and the patient that the data is kept private and secure.

MedcoMD needs this system to satisfy the following use cases:

- Doctors use the system to access Patients' medical records, insurance information, and add new vital sign measurements.
- Receptionists use the system to verify the Patients' identity, using a combination of their contact information and the last four digits of their Social Security Number (SSN).
- Receptionists can view a Patient's insurance policy provider, but not their policy number.
- Receptionists cannot access a Patient's medical records.

MedcoMD is also concerned with disclosure of sensitive data through any of the following methods:


- Accidental disclosure of data on the Receptionist's publicly-viewable screen.
- Direct access to the database by a superuser such as a database administrator.
- Capture of data over an insecure network.
- Access to the data by reading a server's memory.
- Access to the on-disk data by reading database or backup files.



What can MedcoMD do to balance the functionality and access restrictions of their Medical Care Management System?

Comparison of Security Features

The MedcoMD engineers review the Medical Care Management System specification and research the proper solution for limiting access to

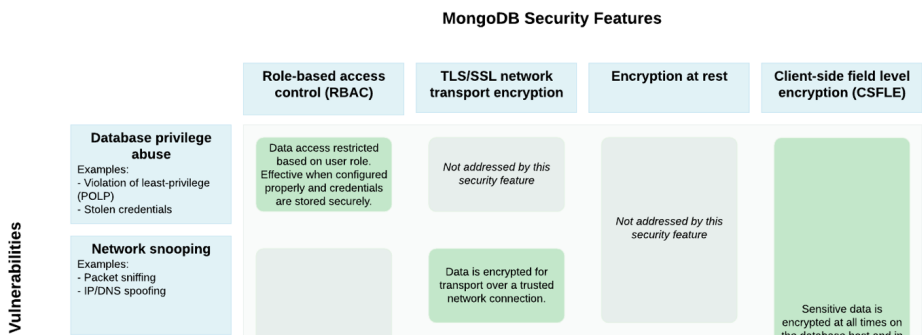
sensitive data.

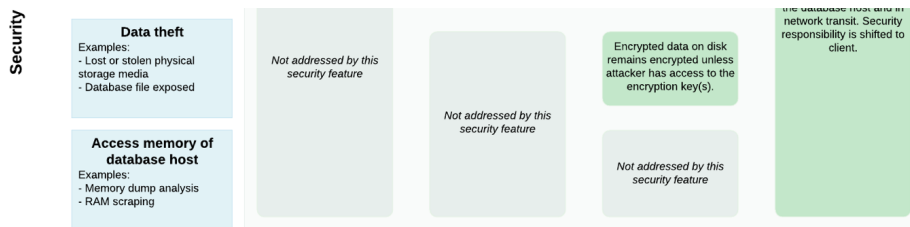
The first MongoDB security feature they evaluated was Role-Based Access Control  which allows administrators to grant and restrict collection-level permissions for users. With the appropriate role definition and assignment, this solution prevents accidental disclosure of data and access. However, it does not prevent capture of the data over an insecure network, direct access of data by a superuser, access to data by reading the server’s memory, or access to on-disk data by reading the database or backup files.

The next MongoDB security features they evaluated were Encryption at Rest  which encrypts the database files on disk and Transport Encryption using TLS/SSL  which encrypts data over the network. When applied together, these two features prevent access to on-disk database files as well as capture of the data on the network, respectively. When combined with Role-Based Access Control, these three security features offer near-comprehensive security coverage of the sensitive data, but lack a mechanism to prevent the data from being read from the server’s memory.

Finally, the MedcoMD engineers discovered a feature that independently satisfies all the security criteria. Client-side Field Level Encryption allows the engineers to specify the fields of a document that should be kept encrypted. Sensitive data is transparently encrypted/decrypted by the client and only communicated to and from the server in encrypted form. This mechanism keeps the specified data fields secure in encrypted form on both the server and the network. While all clients have access to the non-sensitive data fields, only appropriately-configured CSFLE clients are able to read and write the sensitive data fields.

The following diagram is a list of MongoDB security features offered and the potential security vulnerabilities that they address:





MedcoMD will provide Receptionists with a client that is not configured to access data encrypted with CSFLE. This will prevent them from viewing the sensitive fields and accidentally leaving them displayed on-screen in a public area. MedcoMD will provide Doctors with a client with CSFLE enabled which will allow them to access the sensitive data fields in the privacy of their own office.

Equipped with CSFLE, MedcoMD can keep their sensitive data secure and compliant to data privacy regulations with MongoDB.

Implementation

This section explains the following configuration and implementation details of CSFLE:

- Software required to run your client and server in your local development environment.
- Creation and validation of the encryption keys.
- Configuration of the client for automatic field-level encryption.
- Queries, reads, and writes of encrypted fields.

Requirements

MongoDB Server 4.2 Enterprise

- For installation instructions, refer to the Enterprise Edition Installation Tutorials [↗](#).

MongoDB Driver Compatible with CSFLE

- For a list of drivers that support CSFLE, refer to the driver compatibility table [↗](#).

File System Permissions

- The client application or a privileged user needs permissions to start the mongocryptd [↗](#) process on the host.

Additional Dependencies

- Additional dependencies for specific language drivers are required to use CSFLE or run through examples in this guide. To see the list, select the appropriate driver tab below.

Python

Java (Sync)

Node.js

Dependency Name

Description

JDK 8 or later [↗](#)

While the current driver is compatible with older versions of the JDK, the CSFLE feature is only compatible with JDK 8 and later.

libmongocrypt [↗](#)

The libmongocrypt library contains bindings to communicate with the native library that manages the encryption.

A. Create a Master Key

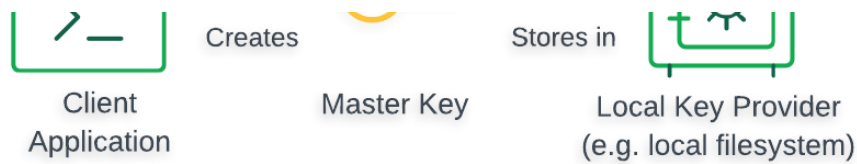
MongoDB Client-Side Field Level Encryption (CSFLE) [↗](#) uses an encryption strategy called *envelope encryption* in which keys used to encrypt/decrypt data (called **data encryption keys**) are encrypted with another key (called the **master key**). For more information on the features of envelope encryption and key management concepts, see [AWS Key Management Service Concepts](#) [↗](#).

In this step, we create and store the master key, used by the MongoDB driver to encrypt data encryption keys, in the **Local Key Provider** which is the filesystem in our local development environment. We refer to this key as the “locally-managed master key” in this guide.

The following diagram shows how the **master key** is created and stored:

Set up Master Key (with local provider)





The **data encryption keys**, generated and used by the MongoDB driver to encrypt and decrypt document fields, are stored in a key vault collection in the same MongoDB replica set as the encrypted data.

LOCAL KEY PROVIDER IS NOT SUITABLE FOR PRODUCTION:

The Local Key Provider is an insecure method of storage and is therefore **not recommended** if you plan to use CSFLE in production. Instead, you should configure a master key in a Key Management System [↗](#) (KMS) which stores and decrypts your data encryption keys remotely.

To learn how to use a KMS in your CSFLE implementation, read the Client-Side Field Level Encryption: Use a KMS to Store the Master Key [↗](#) guide.

To begin development, MedcoMD engineers generate a master key and save it to a file with the **fully runnable code below**:

Python

Java (Sync)

Node.js

The following script generates a 96-byte locally-managed master key and saves it to a file called `master-key.txt` in the directory from which the script is executed.

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.security.SecureRandom;

public class CreateMasterKeyFile {
    public static void main(String[] args) throws IOException {

        byte[] localMasterKey = new byte[96];
        new SecureRandom().nextBytes(localMasterKey);
    }
}
```

```

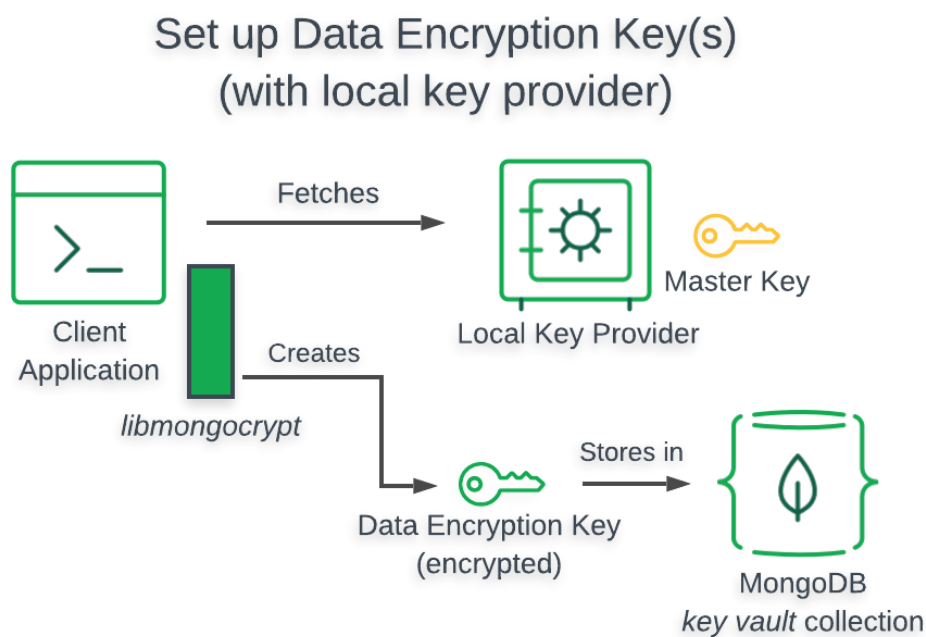
    try (FileOutputStream stream = new FileOutputStream(
        stream.write(localMasterKey);
    }
}
}

```

B. Create a Data Encryption Key

In this section, we generate a data encryption key. The MongoDB driver stores the key in a key vault collection where CSFLE-enabled clients can access the key for automatic encryption and decryption.

The following diagram shows how the **data encryption keys** are created and stored:



The client requires the following configuration values to generate a new data encryption key:

- The locally-managed master key.
- A MongoDB connection string that authenticates on a running server.
- The key vault namespace (database and collection).

Follow the steps below to generate a single data encryption key from the

locally-managed master key.

1 Read the Locally-Managed Master Key from a File

First, retrieve the contents of the master key file that you generated in the Create a Master Key section with the following **code snippet**:

Python

Java (Sync)

Node.js

```
String path = "master-key.txt";

byte[] localMasterKey= new byte[96];

try (FileInputStream fis = new FileInputStream(path)
    fis.readNBytes(localMasterKey, 0, 96);
}
```

NOTE:

The `FileInputStream#readNBytes` [↗](#) method was introduced in Java 9. The helper method is used in this guide to keep the implementation concise. If you are using JDK 8, you may consider implementing a custom solution [↗](#) to read a file into a byte array.

2 Specify KMS Provider Settings

Next, specify the KMS provider settings. The client uses these settings to discover the master key. Set the provider name to `local` when using a local master key in the following **code snippet**:

Python

Java (Sync)

Node.js

The KMS provider settings are stored in a Map in order to use the `kmsProviders` helper method [↗](#) for the `ClientEncryptionSettings` Builder.

```
Map<String, Object> keyMap = new HashMap<String, Obj
```

```
keyMap.put("key", localMasterKey);
```

```
Map<String, Map<String, Object>> kmsProviders = new  
kmsProviders.put("local", keyMap);
```

3 Create a Data Encryption Key

Construct a client with the MongoDB connection string and key vault namespace configuration, and create a data encryption key with the following **code snippet**. The key vault in this example uses the encryption database and `__keyVault` collection.

Python

Java (Sync)

Node.js

```
String connectionString = "mongodb://localhost:27017  
String keyVaultNamespace = "encryption.__keyVault";
```

```
ClientEncryptionSettings clientEncryptionSettings =  
    .keyVaultMongoClientSettings(MongoClientSettings  
        .applyConnectionString(new ConnectionString(  
            .build())  
        .keyVaultNamespace(keyVaultNamespace)  
        .kmsProviders(kmsProviders)  
        .build());
```

```
ClientEncryption clientEncryption = ClientEncryption  
BsonBinary dataKeyId = clientEncryption.createDataKey  
System.out.println("DataKeyId [UUID]: " + dataKeyId.
```

```
String base64DataKeyId = Base64.getEncoder().encodeT  
System.out.println("DataKeyId [base64]: " + base64Da
```

The `createDataKey()` method returns a `BsonBinary` [🔗](#) object from which we can extract the UUID and Base64 representations of the key id.

The `_id` field of the data encryption key is represented as a **UUID** and is encoded in **Base64** format. Use your **Base64**-encoded data key id when specified for the remainder of this guide.

The output from the code above should resemble the following:

```
DataKeyId [UUID]: de4d775a-4499-48bc-bb93-3f81c3c907
DataKeyId [base64]: 3k13WkSZSLy7kwAAP4HDyQ==
```

NOTE:

Ensure that the client has ReadWrite permissions on the specified key vault namespace.

4 Verify that the Data Encryption Key was Created

Query the key vault collection for the data encryption key that was inserted as a document into your MongoDB replica set using the key id printed in the prior step with the following **code snippet**.

Python

Java (Sync)

Node.js

```
String connectionString = "mongodb://localhost:27017
String keyVaultDb = "encryption";
String keyVaultCollection = "__keyVault";
String base64KeyId = "3k13WkSZSLy7kwAAP4HDyQ=="; //

MongoClient mongoClient = MongoClient.create(connec
MongoCollection<Document> collection = mongoClient.g

Bson query = Filters.eq("_id", new Binary((byte) 4,
Document doc = collection
    .find(query)
    .first();

System.out.println(doc);
```

This code example should print a retrieved document that resembles the following:

```
Document{{
  _id=dad3a063-4f9b-48f8-bf4e-7ca9d323fd1c,
```

```
keyMaterial=org.bson.types.Binary@40e1535,  
creationDate=Wed Sep 25 22:22:54 EDT 2019,  
updateDate=Wed Sep 25 22:22:54 EDT 2019,  
status=0,  
masterKey=Document{{provider=local}}  
}}
```

VIEW THE EXTENDED JSON REPRESENTATION OF THE DATA KEY:

While the `Document` class is the `Document` type [↗](#) most commonly used to work with query results, we can use the `BsonDocument` class to view the data key document as extended JSON. Replace the `Document` assignment code with the following to retrieve and print a `BsonDocument`:

```
BsonDocument doc = collection  
    .withDocumentClass(BsonDocument.class)  
    .find(query)  
    .first();  
  
System.out.println(doc);
```

This retrieved document contains the following data:

- Data encryption key id (stored as a UUID).
- Data encryption key in encrypted form.
- KMS provider information for the master key.
- Other metadata such as creation and last modified date.

C. Specify Encrypted Fields Using JSON Schema

MongoDB drivers use an extended version of the JSON Schema [↗](#) standard to configure automatic client-side encryption and decryption of specific fields of the documents in a collection.

NOTE:

Automatic CSFLE requires MongoDB Enterprise or MongoDB Atlas.

The MongoDB CSFLE extended JSON Schema standard requires the following information:

- The encryption algorithm to use when encrypting each field (Deterministic Encryption [↗](#) or Random Encryption [↗](#))
- One or more data encryption keys encrypted with the CSFLE master key
- The BSON Type of each field (only required for deterministically encrypted fields)

CSFLE JSON SCHEMA DOES NOT SUPPORT DOCUMENT VALIDATION:

MongoDB drivers use JSON Schema syntax to specify encrypted fields and *only* support field-level encryption-specific keywords documented in Automatic Encryption JSON Schema Syntax [↗](#). Any other document validation instances will cause the client to throw an error.

SERVER-SIDE JSON SCHEMA:


You can prevent clients that are not configured with the appropriate client-side JSON Schema from writing unencrypted data to a field by using server-side JSON Schema. The server-side JSON Schema provides only supplemental enforcement of the client-side JSON Schema. For more details on server-side document validation implementation, see Enforce Field Level Encryption Schema [↗](#).

The MedcoMD engineers receive specific requirements for the fields of data and their encryption strategies. The following table illustrates the data model of the Medical Care Management System.

Field type	Encryption Algorithm	BSON Type
Name	Non-Encrypted	String
SSN	Deterministic	Int

Blood Type	Random	String
Medical Records	Random	Array
Insurance: Policy Number	Deterministic	Int (embedded inside insurance object)
Insurance: Provider	Non-Encrypted	String (embedded inside insurance object)

Data Encryption Key

The MedcoMD engineers created a single data key to use when encrypting all fields in the data model. To configure this, they specify the encryptMetadata  key at the root level of the JSON Schema. As a result, all encrypted fields defined in the properties field of the schema will inherit this encryption key unless specifically overwritten.

```
{
  "bsonType" : "object",
  "encryptMetadata" : {
    "keyId" : // copy and paste your keyId generated
  },
  "properties": {
    // copy and paste your field schemas here
  }
}
```

MedcoMD engineers create JSON objects for each field and append them to the properties map.

SSN

The ssn field represents the patient's social security number. This field is sensitive and should be encrypted. MedcoMD engineers decide upon deterministic encryption based on the following properties:

- **Quervable**

- High cardinality

```
"ssn": {
  "encrypt": {
    "bsonType": "int",
    "algorithm": "AEAD_AES_256_CBC_HMAC_SHA_512-Deter
  }
}
```

Blood Type

The `bloodType` field represents the patient's blood type. This field is sensitive and should be encrypted. MedcoMD engineers decide upon random encryption based on the following properties:

- No plans to query
- Low cardinality

```
"bloodType": {
  "encrypt": {
    "bsonType": "string",
    "algorithm": "AEAD_AES_256_CBC_HMAC_SHA_512-Randc
  }
}
```

Medical Records

The `medicalRecords` field is an array that contains a set of medical record documents. Each medical record document represents a separate visit and specifies information about the patient at that time, such as their blood pressure, weight, and heart rate. This field is sensitive and should be encrypted. MedcoMD engineers decide upon random encryption based on the following properties:

- Array fields must use random encryption with CSFLE to enable auto-encryption

```
"medicalRecords": [
```

```

    "medicalRecords": {
      "encrypt": {
        "bsonType": "array",
        "algorithm": "AEAD_AES_256_CBC_HMAC_SHA_512-Random"
      }
    }
  }
}

```

Insurance Policy Number

The `insurance.policyNumber` field is embedded inside the `insurance` field and represents the patient's policy number. This policy number is a distinct and sensitive field. MedcoMD engineers decide upon deterministic encryption based on the following properties:

- Queryable
- High cardinality

```

"insurance": {
  "bsonType": "object",
  "properties": {
    "policyNumber": {
      "encrypt": {
        "bsonType": "int",
        "algorithm": "AEAD_AES_256_CBC_HMAC_SHA_512-Deterministic"
      }
    }
  }
}

```

Recap

MedcoMD engineers created a JSON Schema that satisfies their requirements of making sensitive data queryable and secure. View the full JSON Schema for the Medical Care Management System [↗](#).

Python

Java (Sync)

Node.js

View the **complete runnable** helper code in Java [↗](#).

D. Create the MongoDB Client

The MedcoMD engineers now have the JSON Schema and encryption keys necessary to create a CSFLE-enabled MongoDB client.

They build the client to communicate with a MongoDB cluster and perform actions such as securely reading and writing documents with encrypted fields.

About the Mongocryptd Application

The MongoDB client communicates with a separate encryption application called `mongocryptd` which automates the client-side field level encryption. This application is installed with MongoDB Enterprise Server (version 4.2 and later) [↗](#).

When we create a CSFLE-enabled MongoDB client, the `mongocryptd` process is automatically started by default, and handles the following responsibilities:

- Validates the encryption instructions defined in the JSON Schema and flags the referenced fields for encryption in read and write operations.
- Prevents unsupported operations from being executed on encrypted fields.

When the `mongocryptd` process is started with the client driver, you can provide configurable parameters including:

Python	Java (Sync)	Node.js
--------	-------------	---------

Name	Description
port	Listening port. Specify this value in the <code>AutoEncryptionSett</code>

EXAMPLE:

```
List<String> spawnArgs = new Arra
```

```
spawnArgs.add("--port=30000");

Map<String, Object> extraOpts = n
extraOpts.put("mongocryptdSpawnAr

AutoEncryptionSettings autoEncryp
...
.extraOptions(extraOpts);
```



idleShutdownTimeoutSecs

Number of idle seconds in which the mongocryptd process must be started. Specify this value in the AutoEncryptionSettings object.

CLOUD TOOLS
GUIDES

EXAMPLE:

```
List<String> spawnArgs = new ArrayList<>();
spawnArgs.add("--idleShutdownTimeoutSecs=60");

Map<String, Object> extraOpts = new HashMap<>();
extraOpts.put("mongocryptdSpawnArgs", spawnArgs);

AutoEncryptionSettings autoEncryp
...
.extraOptions(extraOpts);
```

Default: 60

NOTE:

If a mongocryptd process is already running on the port specified by the driver, the driver may log a warning and continue to operate without spawning a new process. Any settings specified by the driver only apply once the existing process exits and a new encrypted client attempts to connect.

For additional information on `mongocryptd`, refer to the `mongocryptd` manual page [↗](#).

The MedcoMD engineers use the following procedure to configure and instantiate the MongoDB client:

1 Specify the Key Vault Collection Namespace

The key vault collection contains the data key that the client uses to encrypt and decrypt fields. MedcoMD uses the collection `encryption.__keyVault` as the key vault in the following **code snippet**.

Python

Java (Sync)

Node.js

```
String keyVaultNamespace = "encryption.__keyVault";
```

2 Specify the Local Master Encryption Key

The client expects a key management system to store and provide the application's master encryption key. For now, MedcoMD only has a local master key, so they use the `local` KMS provider and specify the key inline with the following **code snippet**.

Python

Java (Sync)

Node.js

```
Map<String, Object> keyMap = new HashMap<String, Object>();
keyMap.put("key", localMasterKey);

Map<String, Map<String, Object>> kmsProviders = new HashMap<String, Map<String, Object>>();
kmsProviders.put("local", keyMap);
```

3 Map the JSON Schema to the Patients Collection

The MedcoMD engineers assign their schema to a variable. The JSON Schema that MedcoMD defined doesn't explicitly specify the collection to which it applies. To assign the schema, they map it to the `medicalRecords.patients` collection namespace in the following **code snippet**:

Python

Java (Sync)

Node.js

```
HashMap<String, BsonDocument> schemaMap = new HashMa
schemaMap.put("medicalRecords.patients", BsonDocumen
```

4 Specify the Location of the Encryption Binary

MongoDB drivers communicate with the `mongocryptd` encryption binary to perform automatic client-side field level encryption. The `mongocryptd` process performs the following:

- Validates the encryption instructions defined in the JSON Schema and flags the referenced fields for encryption in read and write operations.
- Prevents unsupported operations from being executed on encrypted fields.

Configure the client to spawn the `mongocryptd` process by specifying the path to the binary using the following configuration options:

Python

Java (Sync)

Node.js

```
Map<String, Object> extraOptions = new HashMap<Strin
extraOptions.put("mongocryptdSpawnPath", "/usr/local
```

ENCRYPTION BINARY DAEMON:

If the `mongocryptd` daemon is already running, you can configure the client to skip starting it by passing the following

option:

```
extraOptions.put("mongocryptdBypassSpawn", true);
```

5 Create the MongoClient

To create the CSFLE-enabled client, MedcoMD instantiates a standard MongoDB client object with the additional automatic encryption settings with the following **code snippet**:

Python

Java (Sync)

Node.js

```
MongoClientSettings clientSettings = MongoClientSettings
    .applyConnectionString(new ConnectionString("mon
    .autoEncryptionSettings(AutoEncryptionSettings.b
        .keyVaultNamespace(keyVaultNamespace)
        .kmsProviders(kmsProviders)
        .schemaMap(schemaMap)
        .extraOptions(extraOptions)
        .build())
    .build();
```

```
MongoClient mongoClient = MongoClient.create(client
```

E. Perform Encrypted Read/Write Operations

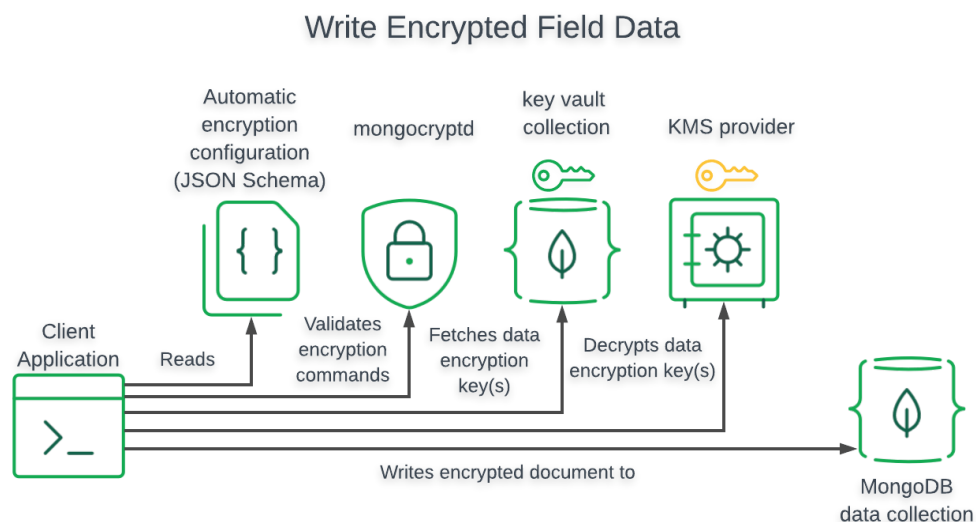
The MedcoMD engineers now have a CSFLE-enabled client and can test that the client can perform queries that meet the requirements.

Doctors should be able to read and write to all fields, and receptionists should only be allowed to read and write to non-sensitive fields.

Insert a Document with Encrypted Fields

The following diagram shows the steps taken by the client application

and driver to perform a write of field-level encrypted data:



MedcoMD engineers write a function to create a new patient record with the following **code snippet**:

Python

Java (Sync)

Node.js

```
public static void insertPatient(
    MongoCollection collection,
    String name,
    int ssn,
    String bloodType,
    ArrayList<Document> medicalRecords,
    int policyNumber,
    String provider
) {

    Document insurance = new Document()
        .append("policyNumber", policyNumber)
        .append("provider", provider);

    Document patient = new Document()
        .append("name", name)
        .append("ssn", ssn)
        .append("bloodType", bloodType)
        .append("medicalRecords", medicalRecords)
        .append("insurance", insurance);
```

```

collection.insertOne(patient);
}

```

When a CSFLE-enabled client inserts a new patient record into the Medical Care Management System, it automatically encrypts the fields specified in the JSON Schema. This operation creates a document similar to the following:

```

{
  "_id": "5d7a7bbe6d58fd263b6d7315",
  "name": "Jon Doe",
  "ssn": "Ac+ZbPM+sk7gl7CJCcIzlRAQUJ+uo/0WhqX+KbTNdhqCs",
  "bloodType": "As+ZbPM+sk7gl7CJCcIzlRACESwHCTCtK/lQV9k",
  "medicalRecords": "As+ZbPM+sk7gl7CJCcIzlRAEFt249toVYC",
  "insurance": {
    "provider": "MaestCare",
    "policyNumber": "Ac+ZbPM+sk7gl7CJCcIzlRAQm7kFhN1k"
  }
}

```

NOTE:

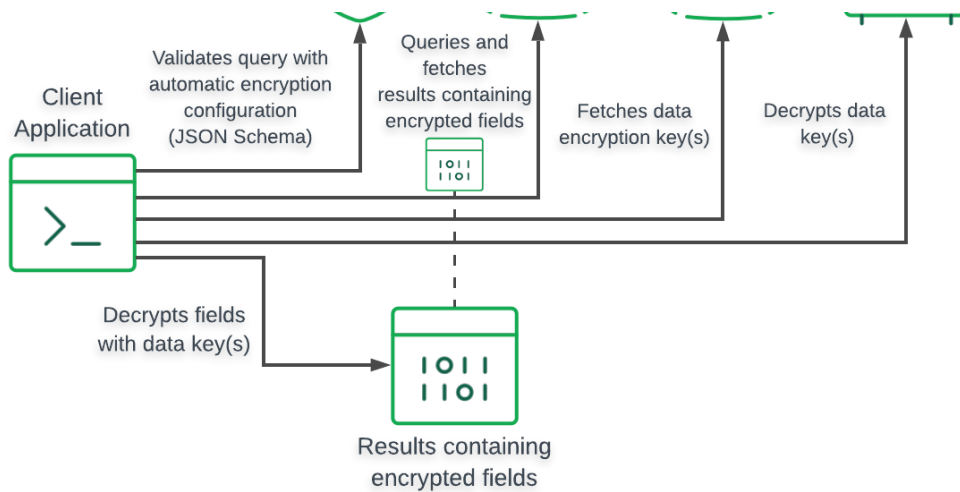
Clients that do not have CSFLE configured will insert unencrypted data. We recommend using server-side schema validation [\[link\]](#) to enforce encrypted writes for fields that should be encrypted.

Query for Documents on a Deterministically Encrypted Field

The following diagram shows the steps taken by the client application and driver to query and decrypt field-level encrypted data:

Read Data From Encrypted Fields





You can run queries on documents with encrypted fields using standard MongoDB driver methods. When a doctor performs a query in the Medical Care Management System to search for a patient by their SSN, the driver decrypts the patient's data before returning it:

```

{
  "_id": "5d6ecdce70401f03b27448fc",
  "name": "Jon Doe",
  "ssn": 241014209,
  "bloodType": "AB+",
  "medicalRecords": [
    {
      "weight": 180,
      "bloodPressure": "120/80"
    }
  ],
  "insurance": {
    "provider": "MaestCare",
    "policyNumber": 123142
  }
}
  
```

NOTE:

For queries using a client that is not configured to use CSFLE, such as when receptionists in the Medical Care Management System search for a patient with their `ssn`, a null value is returned. A client without CSFLE configured cannot query on a sensitive field.

Query for Documents on a Randomly Encrypted Field

WARNING:

You cannot directly query for documents on a randomly encrypted field, however you can use another field to find the document that contains an approximation of the randomly encrypted field data.

MedcoMD engineers determined that the fields they randomly encrypted would not be used to find patients records. Had this been required, for example, if the patient's ssn was randomly encrypted, MedcoMD engineers could have included another plain-text field called `last4ssn` that contains the last 4 digits of the `ssn` field. They could then query on this field as a proxy for the `ssn`.

```
{
  "_id": "5d6ecdce70401f03b27448fc",
  "name": "Jon Doe",
  "ssn": 241014209,
  "last4ssn": 4209,
  "bloodType": "AB+",
  "medicalRecords": [
    {
      "weight": 180,
      "bloodPressure": "120/80"
    }
  ],
  "insurance": {
    "provider": "MaestCare",
    "policyNumber": 123142
  }
}
```

Summary

MedcoMD wanted to develop a system that securely stores sensitive medical records for their patients. They also wanted strong data access and security guarantees that do not rely on individual users. After researching the available options, MedcoMD determined that MongoDB

Client-Side Field Level Encryption satisfies their requirements and decided to implement it in their application. To implement CSFLE they:

1. Created a Locally-Managed Master Encryption Key

A locally-managed master key allowed MedcoMD to rapidly develop the client application without external dependencies and avoid accidentally leaking sensitive production credentials.

2. Generated an Encrypted Data Key with the Master Key

CSFLE uses envelope encryption, so they generated a data key that encrypts and decrypts each field and then encrypted the data key using a master key. This allows MedcoMD to store the encrypted data key in MongoDB so that it is shared with all clients while preventing access to clients that don't have access to the master key.

3. Created a JSON Schema

CSFLE can automatically encrypt and decrypt fields based on a provided JSON Schema that specifies which fields to encrypt and how to encrypt them.

4. Tested and Validated Queries with the CSFLE Client

MedcoMD engineers tested their CSFLE implementation by inserting and querying documents with encrypted fields. They then validated that clients without CSFLE enabled could not read the encrypted data.

Additional Information

Download Example Project

To view and download a runnable example of CSFLE, select your driver below:

Python

Java (Sync)

Node.js

GitHub: Java CSFLE runnable example [↗](#)

Move to Production

In this guide, we stored the master key in your local filesystem. Since your data encryption keys would be readable by anyone that gains direct access to your master key, we **strongly recommend** that you use a more secure storage location such as a Key Management System (KMS).

For more information on securing your master key, see our step-by-step guide to integrating with Amazon KMS [↗](#).

Further Reading

For more information on client-side field level encryption in MongoDB, check out the reference docs in the server manual:

- Client-Side Field Level Encryption [↗](#)
- Automatic Encryption JSON Schema Syntax [↗](#)
- Manage Client-Side Encryption Data Keys [↗](#)

For additional information on MongoDB CSFLE API, see the official Java driver documentation [↗](#)