**DO NOT READ THIS FILE ON GITHUB, GUIDES ARE PUBLISHED ON https://guides.rubyonrails.org.**

# Active Record Encryption

This guide covers encrypting your database information using Active Record.

After reading this guide, you will know:

- How to set up database encryption with Active Record.
- How to migrate unencrypted data
- How to make different encryption schemes coexist
- How to use the API
- How to configure the library and how to extend it

---

Active Record supports application-level encryption. It works by declaring which attributes should be encrypted and seamlessly encrypting and decrypting them when necessary. The encryption layer sits between the database and the application. The application will access unencrypted data, but the database will store it encrypted.

## Why Encrypt Data at the Application Level?

Active Record Encryption exists to protect sensitive information in your application. A typical example is personally identifiable information from users. But why would you want application-level encryption if you are already encrypting your database at rest?

As an immediate practical benefit, encrypting sensitive attributes adds an additional security layer. For example, if an attacker gained access to your database, a snapshot of it, or your application logs, they wouldn't be able to make sense of the encrypted information. Additionally, encryption can prevent developers from unintentionally exposing users' sensitive data in application logs.

But more importantly, by using Active Record Encryption, you define what constitutes sensitive information in your application at the code level. Active Record Encryption enables granular control of data access in your application and services consuming data from your application. For example, consider auditable Rails consoles that protect encrypted data or check the built-in system to filter controller params automatically.

## Basic Usage

### Setup

First, you need to add some keys to your Rails credentials. Run `bin/rails db:encryption:init` to generate a random key set:

```
$ bin/rails db:encryption:init
Add this entry to the credentials of the target environment:

active_record_encryption:
  primary_key: EGY8WhulUOXixybod7ZWwMIL68R9o5kC
  deterministic_key: aPA5XyALhf75NNnMzaspW7akTfZp0lPY
  key_derivation_salt: xEY0dt6TZcAMg52K7O84wYzkjvbA62Hz
```

NOTE: These generated values are 32 bytes in length. If you generate these yourself, the minimum lengths you should use are 12 bytes for the primary key (this will be used to derive the AES 32 bytes key) and 20 bytes for the salt.

### Declaration of Encrypted Attributes

Encryptable attributes are defined at the model level. These are regular Active Record attributes backed by a column with the same name.

```ruby
class Article < ApplicationRecord
  encrypts :title
end
```

The library will transparently encrypt these attributes before saving them in the database and will decrypt them upon retrieval:

```ruby
article = Article.create title: "Encrypt it all!"
article.title # => "Encrypt it all!"
```

But, under the hood, the executed SQL looks like this:

```
INSERT INTO `articles` (`title`) VALUES ('{\"p\":\"n7JO/ol+a7DRMeaE\",\"h\":{\"iv\":\"DXZMDW
```

Because Base 64 encoding and metadata are stored with the values, encryption requires extra space in the column. You can estimate the worst-case overload at around 250 bytes when the built-in envelope encryption key provider is used. This overload is negligible for medium and large text columns, but for `string` columns of 255 bytes, you should increase their limit accordingly (510 bytes is recommended).

### Deterministic and Non-deterministic Encryption

By default, Active Record Encryption uses a non-deterministic approach to encryption. Non-deterministic, in this context, means that encrypting the same content with the same password twice will result in different ciphertexts. This

approach improves security by making crypto-analysis of ciphertexts harder, and querying the database impossible.

You can use the `deterministic:` option to generate initialization vectors in a deterministic way, effectively enabling querying encrypted data.

```
class Author < ApplicationRecord
  encrypts :email, deterministic: true
end

Author.find_by_email("some@email.com") # You can query the model normally
```

The non-deterministic approach is recommended unless you need to query the data.

NOTE: In non-deterministic mode, Active Record uses AES-GCM with a 256-bits key and a random initialization vector. In deterministic mode, it also uses AES-GCM, but the initialization vector is generated as an HMAC-SHA-256 digest of the key and contents to encrypt.

NOTE: You can disable deterministic encryption by omitting a `deterministic_key`.

## Features

### Action Text

You can encrypt action text attributes by passing `encrypted: true` in their declaration.

```
class Message < ApplicationRecord
  has_rich_text :content, encrypted: true
end
```

NOTE: Passing individual encryption options to action text attributes is not supported yet. It will use non-deterministic encryption with the global encryption options configured.

### Fixtures

You can get Rails fixtures encrypted automatically by adding this option to your `test.rb`:

```
config.active_record.encryption.encrypt_fixtures = true
```

When enabled, all the encryptable attributes will be encrypted according to the encryption settings defined in the model.

**Action Text Fixtures** To encrypt action text fixtures, you should place them in `fixtures/action_text/encrypted_rich_texts.yml`.

### Supported Types

`active_record.encryption` will serialize values using the underlying type before encrypting them, but *they must be serializable as strings*. Structured types like `serialized` are supported out of the box.

If you need to support a custom type, the recommended way is using a serialized attribute. The declaration of the serialized attribute should go **before** the encryption declaration:

```
# CORRECT
class Article < ApplicationRecord
  serialize :title, Title
  encrypts :title
end

# INCORRECT
class Article < ApplicationRecord
  encrypts :title
  serialize :title, Title
end
```

### Ignoring Case

You might need to ignore casing when querying deterministically encrypted data. Two approaches make accomplishing this easier:

You can use the `:downcase` option when declaring the encrypted attribute to downcase the content before encryption occurs.

```
class Person
  encrypts :email_address, deterministic: true, downcase: true
end
```

When using `:downcase`, the original case is lost. In some situations, you might want to ignore the case only when querying while also storing the original case. For those situations, you can use the option `:ignore_case`. This requires you to add a new column named `original_<column_name>` to store the content with the case unchanged:

```
class Label
  encrypts :name, deterministic: true, ignore_case: true # the content with the original ca
end
```

### Support for Unencrypted Data

To ease migrations of unencrypted data, the library includes the option `config.active_record.encryption.support_unencrypted_data`. When set to `true`:

- Trying to read encrypted attributes that are not encrypted will work normally, without raising any error
- Queries with deterministically-encrypted attributes will include the "clear text" version of them to support finding both encrypted and unencrypted content. You need to set `config.active_record.encryption.extend_queries = true` to enable this.

**This option is meant to be used during transition periods** while clear data and encrypted data must coexist. Both are set to `false` by default, which is the recommended goal for any application: errors will be raised when working with unencrypted data.

### Support for Previous Encryption Schemes

Changing encryption properties of attributes can break existing data. For example, imagine you want to make a deterministic attribute non-deterministic. If you just change the declaration in the model, reading existing ciphertexts will fail because the encryption method is different now.

To support these situations, you can declare previous encryption schemes that will be used in two scenarios:

- When reading encrypted data, Active Record Encryption will try previous encryption schemes if the current scheme doesn't work.
- When querying deterministic data, it will add ciphertexts using previous schemes so that queries work seamlessly with data encrypted with different schemes. You must set `config.active_record.encryption.extend_queries = true` to enable this.

You can configure previous encryption schemes:

- Globally
- On a per-attribute basis

**Global Previous Encryption Schemes** You can add previous encryption schemes by adding them as list of properties using the `previous` config property in your `application.rb`:

```
config.active_record.encryption.previous = [ { key_provider: MyOldKeyProvider.new } ]
```

**Per-attribute Encryption Schemes** Use `:previous` when declaring the attribute:

```
class Article
  encrypts :title, deterministic: true, previous: { deterministic: false }
end
```

**Encryption Schemes and Deterministic Attributes** When adding previous encryption schemes:

- With **non-deterministic encryption**, new information will always be encrypted with the *newest* (current) encryption scheme.
- With **deterministic encryption**, new information will always be encrypted with the *oldest* encryption scheme by default.

Typically, with deterministic encryption, you want ciphertexts to remain constant. You can change this behavior by setting `deterministic: { fixed: false }`. In that case, it will use the *newest* encryption scheme for encrypting new data.

### Unique Constraints

NOTE: Unique constraints can only be used with deterministically encrypted data.

**Unique Validations** Unique validations are supported normally as long as extended queries are enabled (`config.active_record.encryption.extend_queries = true`).

```
class Person
  validates :email_address, uniqueness: true
  encrypts :email_address, deterministic: true, downcase: true
end
```

They will also work when combining encrypted and unencrypted data,git and when configuring previous encryption schemes.

NOTE: If you want to ignore case, make sure to use `downcase:` or `ignore_case:` in the `encrypts` declaration. Using the `case_sensitive:` option in the validation won't work.

**Unique Indexes** To support unique indexes on deterministically-encrypted columns, you need to ensure their ciphertext doesn't ever change.

To encourage this, deterministic attributes will always use the oldest available encryption scheme by default when multiple encryption schemes are configured. Otherwise, it's your job to ensure encryption properties don't change for these attributes, or the unique indexes won't work.

```
class Person
  encrypts :email_address, deterministic: true
end
```

### Filtering Params Named as Encrypted Columns

By default, encrypted columns are configured to be automatically filtered in Rails logs. You can disable this behavior by adding the following to your `application.rb`:

When generating the filter parameter, it will use the model name as a prefix. E.g: For `Person#name` the filter parameter will be `person.name`.

```
config.active_record.encryption.add_to_filter_parameters = false
```

In case you want exclude specific columns from this automatic filtering, add them to `config.active_record.encryption.excluded_from_filter_parameters`.

### Encoding

The library will preserve the encoding for string values encrypted non-deterministically.

Because encoding is stored along with the encrypted payload, values encrypted deterministically will force UTF-8 encoding by default. Therefore the same value with a different encoding will result in a different ciphertext when encrypted. You usually want to avoid this to keep queries and uniqueness constraints working, so the library will perform the conversion automatically on your behalf.

You can configure the desired default encoding for deterministic encryption with:

```
config.active_record.encryption.forced_encoding_for_deterministic_encryption = Encoding::US
```

And you can disable this behavior and preserve the encoding in all cases with:

```
config.active_record.encryption.forced_encoding_for_deterministic_encryption = nil
```

## Key Management

Key providers implement key management strategies. You can configure key providers globally, or on a per attribute basis.

### Built-in Key Providers

**DerivedSecretKeyProvider** A key provider that will serve keys derived from the provided passwords using PBKDF2.

```
config.active_record.encryption.key_provider = ActiveRecord::Encryption::DerivedSecretKeyPro
```

NOTE: By default, `active_record.encryption` configures a `DerivedSecretKeyProvider` with the keys defined in `active_record.encryption.primary_key`.

**EnvelopeEncryptionKeyProvider** Implements a simple envelope encryption strategy:

- It generates a random key for each data-encryption operation
- It stores the data-key with the data itself, encrypted with a primary key defined in the credential `active_record.encryption.primary_key`.

You can configure Active Record to use this key provider by adding this to your `application.rb`:

```
config.active_record.encryption.key_provider = ActiveRecord::Encryption::EnvelopeEncryptionK
```

As with other built-in key providers, you can provide a list of primary keys in `active_record.encryption.primary_key` to implement key-rotation schemes.

### Custom Key Providers

For more advanced key-management schemes, you can configure a custom key provider in an initializer:

```
ActiveRecord::Encryption.key_provider = MyKeyProvider.new
```

A key provider must implement this interface:

```
class MyKeyProvider
  def encryption_key
  end

  def decryption_keys(encrypted_message)
  end
end
```

Both methods return `ActiveRecord::Encryption::Key` objects:

- `encryption_key` returns the key used for encrypting some content
- `decryption keys` returns a list of potential keys for decrypting a given message

A key can include arbitrary tags that will be stored unencrypted with the message. You can use `ActiveRecord::Encryption::Message#headers` to examine those values when decrypting.

### Model-specific Key Providers

You can configure a key provider on a per-class basis with the `:key_provider` option:

```
class Article < ApplicationRecord
  encrypts :summary, key_provider: ArticleKeyProvider.new
end
```

### Model-specific Keys

You can configure a given key on a per-class basis with the `:key` option:

```
class Article < ApplicationRecord
  encrypts :summary, key: "some secret key for article summaries"
end
```

Active Record uses the key to derive the key used to encrypt and decrypt the data.

### Rotating Keys

`active_record.encryption` can work with lists of keys to support implementing key-rotation schemes:

- The **last key** will be used for encrypting new content.
- All the keys will be tried when decrypting content until one works.

```
active_record
  encryption:
    primary_key:
        - a1cc4d7b9f420e40a337b9e68c5ecec6 # Previous keys can still decrypt existing conte
        - bc17e7b413fd4720716a7633027f8cc4 # Active, encrypts new content
    key_derivation_salt: a3226b97b3b2f8372d1fc6d497a0c0d3
```

This enables workflows in which you keep a short list of keys by adding new keys, re-encrypting content, and deleting old keys.

NOTE: Rotating keys is not currently supported for deterministic encryption.

NOTE: Active Record Encryption doesn't provide automatic management of key rotation processes yet. All the pieces are there, but this hasn't been implemented yet.

### Storing Key References

You can configure `active_record.encryption.store_key_references` to make `active_record.encryption` store a reference to the encryption key in the encrypted message itself.

```
config.active_record.encryption.store_key_references = true
```

Doing so makes for more performant decryption because the system can now locate keys directly instead of trying lists of keys. The price to pay is storage: encrypted data will be a bit bigger.

## API

### Basic API

ActiveRecord encryption is meant to be used declaratively, but it offers an API for advanced usage scenarios.

### Encrypt and Decrypt

```
article.encrypt # encrypt or re-encrypt all the encryptable attributes
article.decrypt # decrypt all the encryptable attributes
```

### Read Ciphertext

```
article.ciphertext_for(:title)
```

#### Check if Attribute is Encrypted or Not

```
article.encrypted_attribute?(:title)
```

## Configuration

### Configuration Options

You can configure Active Record Encryption options in your `application.rb` (most common scenario) or in a specific environment config file `config/environments/<env name>.rb` if you want to set them on a per-environment basis.

WARNING: It's recommended to use Rails built-in credentials support to store keys. If you prefer to set them manually via config properties, make sure you don't commit them with your code (e.g. use environment variables).

**`config.active_record.encryption.support_unencrypted_data`** When true, unencrypted data can be read normally. When false, it will raise errors. Default: `false`.

**`config.active_record.encryption.extend_queries`** When true, queries referencing deterministically encrypted attributes will be modified to include additional values if needed. Those additional values will be the clean version of the value (when `config.active_record.encryption.support_unencrypted_data` is true) and values encrypted with previous encryption schemes, if any (as provided with the `previous:` option). Default: `false` (experimental).

**`config.active_record.encryption.encrypt_fixtures`** When true, encryptable attributes in fixtures will be automatically encrypted when loaded. Default: `false`.

**`config.active_record.encryption.store_key_references`** When true, a reference to the encryption key is stored in the headers of the encrypted message. This makes for faster decryption when multiple keys are in use. Default: `false`.

**`config.active_record.encryption.add_to_filter_parameters`** When true, encrypted attribute names are added automatically to `config.filter_parameters` and won't be shown in logs. Default: `true`.

**`config.active_record.encryption.excluded_from_filter_parameters`** You can configure a list of params that won't be filtered out when `config.active_record.encryption.add_to_filter_parameters` is true. Default: `[]`.

**`config.active_record.encryption.validate_column_size`** Adds a validation based on the column size. This is recommended to prevent storing huge values using highly compressible payloads. Default: `true`.

`config.active_record.encryption.primary_key` The key or lists of keys used to derive root data-encryption keys. The way they are used depends on the key provider configured. It's preferred to configure it via the `active_record_encryption.primary_key` credential.

`config.active_record.encryption.deterministic_key` The key or list of keys used for deterministic encryption. It's preferred to configure it via the `active_record_encryption.deterministic_key` credential.

`config.active_record.encryption.key_derivation_salt` The salt used when deriving keys. It's preferred to configure it via the `active_record_encryption.key_derivation_salt` credential.

`config.active_record.encryption.forced_encoding_for_deterministic_encryption` The default encoding for attributes encrypted deterministically. You can disable forced encoding by setting this option to `nil`. It's `Encoding::UTF_8` by default.

### Encryption Contexts

An encryption context defines the encryption components that are used in a given moment. There is a default encryption context based on your global configuration, but you can configure a custom context for a given attribute or when running a specific block of code.

NOTE: Encryption contexts are a flexible but advanced configuration mechanism. Most users should not have to care about them.

The main components of encryption contexts are:

- `encryptor`: exposes the internal API for encrypting and decrypting data. It interacts with a `key_provider` to build encrypted messages and deal with their serialization. The encryption/decryption itself is done by the `cipher` and the serialization by `message_serializer`.
- `cipher`: the encryption algorithm itself (AES 256 GCM)
- `key_provider`: serves encryption and decryption keys.
- `message_serializer`: serializes and deserializes encrypted payloads (`Message`).

NOTE: If you decide to build your own `message_serializer`, it's important to use safe mechanisms that can't deserialize arbitrary objects. A common supported scenario is encrypting existing unencrypted data. An attacker can leverage this to enter a tampered payload before encryption takes place and perform RCE attacks. This means custom serializers should avoid `Marshal`, `YAML.load` (use `YAML.safe_load` instead), or `JSON.load` (use `JSON.parse` instead).

**Global Encryption Context** The global encryption context is the one used by default and is configured as other configuration properties in your

`application.rb` or environment config files.

```
config.active_record.encryption.key_provider = ActiveRecord::Encryption::EnvelopeEncryptionK
config.active_record.encryption.encryptor = MyEncryptor.new
```

**Per-attribute Encryption Contexts** You can override encryption context params by passing them in the attribute declaration:

```
class Attribute
  encrypts :title, encryptor: MyAttributeEncryptor.new
end
```

**Encryption Context When Running a Block of Code** You can use `ActiveRecord::Encryption.with_encryption_context` to set an encryption context for a given block of code:

```
ActiveRecord::Encryption.with_encryption_context(encryptor: ActiveRecord::Encryption::NullE
  ...
end
```

**Built-in Encryption Contexts**

**Disable Encryption** You can run code without encryption:

```
ActiveRecord::Encryption.without_encryption do
  ...
end
```

This means that reading encrypted text will return the ciphertext, and saved content will be stored unencrypted.

**Protect Encrypted Data** You can run code without encryption but prevent overwriting encrypted content:

```
ActiveRecord::Encryption.protecting_encrypted_data do
  ...
end
```

This can be handy if you want to protect encrypted data while still running arbitrary code against it (e.g. in a Rails console).