AxonIQ GDPR Module Guide

Version 1.2, April 4, 2018

Table of Contents

Chapter 1. Introduction

1.1. About

The core capability of the AxonIQ GDPR Module is to encrypt and decrypt individual fields of a Java object, using a key which is determined by the value of another field of the object. Although this capability may have various different use cases, the main use case foreseen when designing the module was to provide the option to later effectively delete the contents of the encrypted fields, by destroying the key used to encrypt them.

This use case is important because of the potential problem combining the event sourcing architecture pattern with mandatory erasure. With event sourcing, all data is stored as a series of immutable, undeletable events. At first, this seems to preclude erasing any data. By encrypting parts of that data, and storing a key outside of the event stream so that it can be deleted, this becomes possible again: deleting the key will effectively delete the data Of course, other approaches are possible (including changing the immutable events anyway), but these suffer from significant drawbacks and potential operations costs.

Since the majority of Axon Framework-based application use event sourcing, this issue has been a specific concern to AxonIQ as the company behind Axon Framework, and this triggered the creation of this module. Also, the upcoming EU's General Data Protection Regulation (GDPR) is one very important cause of mandatory erasure, since it specifically describes the data subject's 'right to be forgotten'. This has given the module its name. Nevertheless, the module can be used more broadly, both outside Axon Framework, outside event sourcing, and outside the context of GDPR.

1.2. API overview

The most important design goal of the module was to implement the described functionality while minimizing the impact on existing applications. Business logic code must not be affected at all, and data model code should be affected as little as possible.

The core idea to achieve this is to annotate data classes with the encryption requirements, and then set up some infrastructure to ensure that the required encryption takes place transparently to the rest of the application. On the annotation side, the minimum is to tell the module which field determines the key to be used (referred to as the data subject id), and which fields need to be encrypted. A simple example is given below, illustrating the use of the `@DataSubjectId` and `@PersonalData` annotations.

```
1 public class NewCustomerEvent {
2     @DataSubjectId
3     private int id;
4
5     @PersonalData
6     private String name;
7 }
```

Now, encryption and decryption of `NewCustomerEvent` objects can be performed by invoking the appropriate method on the GDPR Module's `FieldEncrypter` class. When this happens, the GDPR Module will inspect the class's annotations and act accordingly. When encrypting, it will look at the value of `id` and then retrieve the existing key with that id or create a new one if it doesn't exist yet. Then, it will encrypt the value of `name` and store this encrypted version in the same field. Because of this architecture, no other changes are necessary.

If your application uses Axon Framework, this process can be even further simplified. The GDPR Module's `FieldEncryptingSerializer` is a wrapper around any other Axon Framework `Serializer` which performs encryption before serializing and decryption after deserializing. By simply registering this new serializer in the Axon Framework configuration, required encryption and decryption will always be performed transparently to the rest of the application. This concept is illustrated in Figure 1 below.

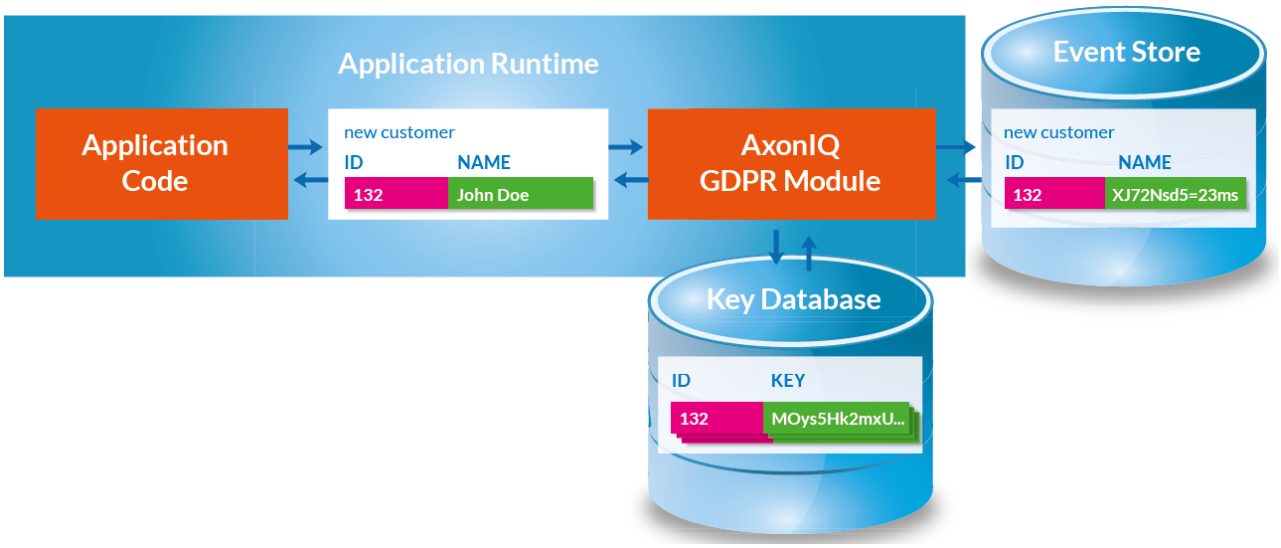

Figure 1. GDPR Module integration

1.3. About this guide

In Chapter 2 we'll discuss the basics of installing the module and running some unit tests. In Chapter 3 we'll discuss all available GDPR Module annotations in detail. This will allow to go beyond the simple case of encrypting single `String` fields. In section Chapter 4 we'll discuss the integration of the module in your application in more detail, including various ways to set up a key store. Lastly, Chapter 5 has some technical background on how the cryptographic algorithms used actually work.

Chapter 2. Getting started

2.1. Software and license file

The GDPR Module is commercial software provided by AxonIQ B.V. Distribution of the software takes place through our download site https://download.axoniq.io. Access requires a username/password. To use the software, you will need the actual software distribution as well as a personalized license file (called `axoniq.license`). If you don't have access to the software yet, please contact sales@axoniq.io. For any technical questions about the module, bug reports and improvement requests, please contact support@axoniq.io.

The license file will be automatically picked up if its in the working directory of the process and called `axoniq.license`. Otherwise, a file location may be specified through the `axoniq.gdpr.license` system property. If the license file cannot be found on the filesystem, the module will try to find the license file as a resource on the classpath. This allows for easier bundling in deployments.

2.2. JCE Unlimited Strength Jurisdiction Policy Files

The GDPR Module using the industry-standard Advanced Encryption Standard (AES) as its core encryption algorithm. AES can be used with 3 different key sizes: 128, 192 and 256 bits. The GDPR Module supports all sizes, but we recommend using its default 256 bits. Section 5.2 has some more information on the background of this.

However, because of legal restrictions on the export of cryptography from the United States to certain countries, the standard JDK/JRE doesn't allow the use of AES key sizes larger than 128 bits. To use the GDPR Module at its default settings, and to run the unit tests, you will need to update your JDK/JRE to support AES-256. The method to do this is to download and install the so-called 'JCE Unlimited Strength Jurisdiction Policy Files' for your JDK/JRE. For example, for Oracle Java 8 they can be found at http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html.

2.3. Versions

The AxonIQ GDPR Module ships in 3 different versions: axon3, axon2 and core. The background of this is in the relation between the module and serializers.

The GDPR Module interacts with serialization in 3 different ways:

- The module itself implements a serializer in `FieldEncryptingSerializer`. This enables it to be transparently included in an Axon Framework application.
- The core `FieldEncrypter` class encrypts objects fields in-place and thus modifies the objects. This is not

expected behaviour from a serializer. For this reason, the `FieldEncryptingSerializer` performs a clone first before performing encryption. (This behaviour can be disabled as a performance optimalization, if desired.) The clone operation is implemented by default by serializing and then deserializing an object, and therefore requires a serializer. (Although this again can be overridden by a different cloning algorithm.)

- The module can encrypt `byte[]`/`String` fields directly into `byte[]`/`String` form (including any collections and arrays of such objects), but that's not true of arbitrary other data types. For those data types, it needs to serialize the object first before it can encrypt.

There are different serialization strategies possible, including plain old `java.io.Serializable`, or serialization to JSON or to XML. Axon Framework has a great abstraction of a serializer which allows all these choices to be plugged in. However, it is slightly different between Axon versions and we also wanted to support using the module without Axon. For this reason, there are 3 different versions of the module:

- The **axon3** version of the module has a dependency on axon-core 3.0.6 and uses the Axon 3 serialization mechanism.
- The **axon2** version of the module has a dependency on axon-core 2.4.6 and uses the Axon 2 serialization mechanism.
- The **core** version of the module doesn't depend on Axon Framework at all. Instead, it contains a copy of the Axon 3 serialization classes (moved to the new package `io.axoniq.gdpr.serialization`), stripped of all things that are specific to Axon framework messaging.

All versions of the module require a Java 8+ JDK.

All versions of the module can optionally be used in combination with Scala, and are compatible with Scala 2.11 and Scala 2.12.

2.4. Installation

The AxonIQ GDPR Module is shipped as an archive containing this documentation and a set of other components for each of the 3 versions described above. The first installation step would be to unpack this archive.

Per version, there is a single jar file to be put on your application's class path. The vast majority of users will use Maven or another build tool that uses the Maven infrastructure. For this reason, we also deliver an accompanying `pom.xml` and and `install.sh`/`.bat` to install the module in a local Maven repository. Of course, in an enterprise situation you may wish to install the module into an enterprise repository system like Nexus as well. Alongside the jar itself, there's a javadoc jar (for Maven) and a prefix that contains the javadoc in prefix format for direct access through a browser.

For each version of the module, there is a subdirectory 'test' with a unit-test project for that version. You may use this both to check your installation and as a source of code samples. To successfully run the unit tests, the following steps are required:

- Install the JCE Unlimited Strength Jurisdiction Policy Files.
- Install the module into your Maven repository with `install.sh/.bat` or otherwise.
- Copy your `axoniq.license` file into the test project prefix.
- Run `mvn verify`

If this works, you can start integrating the GDPR Module with your own application. The first step for doing this would be to add the required dependency to your application,

```
<dependency>
    <groupId>io.axoniq</groupId>
    <artifactId>axoniq-gdpr-XXXX</artifactId>
    <version>YYYY</version>
</dependency>
```

where XXXX is one of `axon3`, `axon2` or `core`, and YYYY is currently `1.2`.

Please note that the `axon3` and `axon2` versions only have an *optional* dependency on `axon-core` which will not be transitively included. The reason is that the GDPR Module should not overrule the Axon Framework version set for the rest of the project. If you use the `axon3` or `axon2` version of the module, the project must also include a dependency on `axon-core` for that major version.

Chapter 3. Annotations

3.1. The `@DataSubjectId` annotation

This field annotation is used to mark the field that identifies the key to be used for encrypting other fields. In its simplest form, it can be used like this:

```
1 @DataSubjectId
2 private UUID id;
```

The GDPR Module will derive a key identifier from the field's value by invoking the `toString()` method, which means that key values should have a meaningful, deterministic implementation of `toString()`. This is already the case for `@AggregateIdentifier` fields in Axon Framework applications. The GDPR Module doesn't try to enforce this restriction though, since there isn't a sufficiently reliable way of doing so. A field annotated with `@DataSubjectId` should not hold a `null` value upon encryption/decryption - this will cause an exception.

In some cases, you may need to use a key identifier which is not present as a field value. For these cases, it is possible to include those key identifiers in the call to `FieldEncrypter`. This is further explained in Section 4.2.1.

3.1.1. The `group` attribute

The `@DataSubjectId` annotation has an optional `String` attribute `group`. This can be used to define multiple keys within the same object, which would allow later on to selectively delete individual (groups of) fields. For instance, a class could look like this:

```
1  public class NewCustomerEvent {
2      @DataSubjectId(group = "name")
3      private int idForName;
4
5      @PersonalData(group = "name")
6      private String name;
7
8      @DataSubjectId(group = "address")
9      private int idForAddress;
10
11     @PersonalData(group = "address")
12     private String address;
13 }
```

This would now lead to two different keys (assuming `idForName` and `idForAddress` hold different values). These keys can be deleted individually, effectively deleting the `name` and `address` fields individually. Of

course, there may only be one natural id field. In the next section, we'll see an elegant solution for this situation using the `prefix` attribute.

If `group` is not specified, it defaults to the group identified by the empty string.

3.1.2. The `prefix` attribute

The `@DataSubjectId` annotation has an optional `String` attribute `prefix`. This value of this attribute will be used to determine the id of the key in the keystore. The actual key id will consist of the value of the field prefixed by the value of `prefix`. If not specified, this defaults to the empty string.

As an illustration, see this example:

```
1 @DataSubjectId(prefix = "aaa")
2 private int id;
```

If the field `id` holds a value of 4, the derived key id will be `aaa4`.

This may very well be combined with the `group` attribute described earlier. The name/address segregation could be achieved like this:

```
1  public class NewCustomerEvent {
2      @DataSubjectId(group = "name", prefix = "name-")
3      @DataSubjectId(group = "address", prefix = "address-")
4      private int id;
5
6      @PersonalData(group = "name")
7      private String name;
8
9      @PersonalData(group = "address")
10     private String address;
11 }
```

Now, if the value of `id` is `913`, there will be 2 different keys in key store, with ids `name-913` and `address-913`. They can be deleted individually to delete individual fields.

3.1.3. Map fields and the `scope` attribute

The `@DataSubjectId` annotation has an optional attribute `scope` of the type `io.axoniq.gdpr.api.Scope`. This is an enum with values `DEFAULT`, `KEY`, `VALUE`, and `BOTH`. If the attribute isn't specified, it will be `DEFAULT`.

For a field type that doesn't implement the `java.util.Map` interface, the value must always be `DEFAULT`. For a field type that does implement the `java.util.Map` interface, the only allowed value is `KEY`. This has the effect that the value of any map key will be treated as a `@DataSubjectId` field for the purpose of processing

the corresponding map value.

3.1.4. Limitations and inheritance

The value of the key identifier in a particular group must always be unambiguous. Therefore, there can be no two `@DataSubjectId` annotations with the same `group` attribute in the same class. The exception to this rule is in `Map` fields where the annotation is applied with `KEY` scope. In this case, the annotation on the map key gets precedence over the annotation on another field when processing the associated map value, which removes the ambiguity. Of course, on the same map field, there can't more than one `@DataSubjectId` annotation with the same `group`.

In a class hierarchy, a subclass may have `@DataSubjectId` fields with the same group as used in the superclass. In this case, the deepest annotation will 'override' the superclass behaviour and thus take precedence.

Although more than one `@DataSubjectId` annotation may be present on the same field assuming they have different `group` attributes, the `@DataSubjectId` annotation cannot be combined with any other GDPR Module annotation with the same `scope` on the same field.

3.2. The `@PersonalData` annotation

This annotation identifies fields to be wholly encrypted by the GDPR Module. It can be applied on those fields where the module can store the encrypted value in the same Java type as the clear text value. Because of this, it doesn't need any secondary place for storage. We call this concept *type-preserving encryption*. Currently, the module supports the following types for `@PersonalData`:

1. `byte[]`
2. `String`
3. An array with a supported type as its component type.
4. Any class implementing `java.util.Collection` or `scala.collection.Iterable` with a supported type as the type parameter, or a supported type as an upper bound in a wildcard type parameter.
5. Any class implementing `java.util.Map` or `scala.collection.Map` with a supported type as the `Map` type parameter for the scope of the annotation (`KEY`, `VALUE` or `BOTH`), or again a supported type as an upper bound in a wildcard type parameter.

Please note that rules 3 - 5 work recursively, so for instance the following is legal:

```
1 @PersonalData
2 List<? extends Set<String[]>> b;
```

When directly encrypting a `String` or `byte[]` field annotated with `@PersonalData`, the GDPR Module will

replace the value of the field. When encrypting an array, `Collection`, or `scala.collection.Iterable`, a few different cases may occur.

If as a result of the operation, the array/`Collection`/`Iterable` doesn't have to be changed itself, it will be left as-is. This may be the case if it's empty, if it was already encrypted when trying encryption, or when its elements are again collections which can be modified without being recreated.

If the collection does have to be changed, there is a different strategy for Java and Scala:

Java

Java collections are generally mutable. If needed, the `FieldEncrypter` will clear the Java collection and then add the new encrypted elements (thus preserving order). In some cases, Java collections are immutable, such as the ones returned by `Collections.unmodifiableList`. By default, the module will try to modify these collections anyhow by performing reflection and directly modifying the internals. This is configurable behavior, which you may turn off by calling `setModifyImmutableCollections(false)` on the `FieldEncrypter`.

Scala

Scala has a more systematic hierarchy of mutable and immutable collections. If a collection is mutable (specifically if it implements `scala.collection.generic.Growable`), the module will clear the collection and add the new encrypted elements, like in the Java case. If a collection does not implement this interface, it is assumed immutable, and the GDPR Module will create a new copy of the collection by calling the `newBuilder()` method on the original collection. The value of the `modifyImmutableCollections` property doesn't play a role when processing Scala collections.

3.2.1. The `group` attribute

The `@PersonalData` annotation has an optional `group` attribute which defaults to the empty string. Please see Section 3.1.1 for a detailed discussion of this attribute.

3.2.2. The `scope` attribute

The `@PersonalData` annotation has an optional attribute `scope` of the type `io.axoniq.gdpr.api.Scope`. This is an enum with values `DEFAULT`, `KEY`, `VALUE`, and `BOTH`. If the attribute isn't specified, it will be `DEFAULT`.

For a field type that doesn't implement the `java.util.Map` or `scala.collection.Map` interface, the value must always be `DEFAULT`. For a field type that does implement the `java.util.Map` or `scala.collection.Map` interface, it must always be something other than `DEFAULT`. When used on a `Map`, the value of this attribute determines which part of the map entry (key, value or both) get encrypted.

3.2.3. The `replacement` attribute

The `@PersonalData` annotation has an optional `replacement` attribute which defaults to the empty string. This attribute is intended to control the replacement value, i.e. the value that the module will put in field if it is deleted (because it notices upon decryption that the required key is no longer available).

When `@PersonalData` is used on a `String` field, the default `ReplacementValueProvider` will use the value of the `replacement` attribute in this case. On all other field types, it will be ignored by default, but it can be used in some way by a custom `ReplacementValueProvider`. Section 4.4 has some examples of this.

3.2.4. Multiple `@PersonalData` annotations and the `reencrypt` attribute

Normally speaking, `@PersonalData` is used under the following two assumptions:

1. There is only one `@PersonalData` annotation per field (or per Map-side in case of a Map-field).
2. Encryption and decryption are *idempotent*: encrypting something which is already encrypted doesn't change the data, and vice versa.

For the main GDPR Module use case, cryptographic erasure, this is fine. The idempotency property makes implementation and data migration easier since a data store can contain mixed encrypted/non-encrypted data without problems.

We've identified a use case where the assumption don't hold and different behaviour is needed. This is the case when the GDPR Module is used in two roles at the same time: to enable cryptographic erasure *and* to encrypt sensitive personal data to protect confidentiality. In this case, we may want to encrypt first with a key to protect confidentiality, and then another time to enable cryptographic erasure of the encrypted data. We use two `@PersonalData` annotations. The following rules apply:

- During encryption, annotations are processed from right to left (so the annotation closest to the field is processed first). During decryption, this is reversed.
- To make sure that the second round of encryption is actually executed, the `reencrypt` attribute must be set to `true`. This disables the check that is normally done to implement the idempotency on encryption. Decryption is unaffected.

As an example, this could be used as follows:

```
class NewPersonCreated {
    @DataSubjectId
    UUID id;

    @PersonalData
    String name;

    @PersonalData(reencrypt = true)
    @PersonalData(group = "sensitive")
    String somethingSecret;
}
```

When processing this class, first, the value of `somethingSecret` will be encrypted with the key belong to group `sensitive`. (Which would have to be pre-provided since we didn't map it to a `@DataSubjectId` - see Section 4.2.1.) After that, the `name` and the already encrypted form of `somethingSecret` will be encrypted in the default group, using the key identified by `id`. Now, deleting key `id` is sufficient to enable cryptographic erasure of all fields. To get access to the value of `somethingSecret`, the user must also have access to the `sensitive` key.

Instead of performing all encrypting operations at the same time, the module can be used to perform the operations on a subset of the groups. See Section 4.2.2 for details.

3.3. The `@SerializedPersonalData` annotation

This annotation identifies fields to be wholly encrypted by the GDPR Module. It can be applied on fields where type-preserving encryption isn't possible and therefore the `@PersonalData` annotation is not allowed. If this annotation is present, the GDPR Module will use the following procedure for encryption:

1. The value of the field will be serialized using the configured serializer.
2. This serialized value gets encrypted.
3. The encrypted value gets stored in a separate *storage* field, which must be `String` or `byte[]` typed and by default has the name of the value field suffixed by 'Encrypted'.

For instance, a `LocalDate` field might be encrypted like this:

```
@SerializedPersonalData
LocalDate dateOfBirth;
byte[] dateOfBirthEncrypted;
```

Compared to `@PersonalData`, this has the advantage that we can now encrypt any field type, however it comes at the cost of having to introduce an additional field, and it incurs some additional CPU and storage overhead.

A difference with `@PersonalData` and `@DeepPersonalData` is that a `@SerializedPersonalData` field will never be examined recursively by going into the elements of an array, collection or map. It will always be encrypted entirely regardless of its type. This also creates a difference in what exactly is erasable: if you use `@PersonalData` or `@DeepPersonalData` on a collection, the individual elements are encrypted and therefore erasable, but the *number* of elements is in the clear. By using `@SerializedPersonalData`, you may obscure this as well (even though the length of the resulting ciphertext may still provide an indication).

3.3.1. The `group` attribute

The `@SerializedPersonalData` annotation has an optional `group` attribute which defaults to the empty string. Please see Section 3.1.1 for a detailed discussion of this attribute.

3.3.2. The `replacement` attribute

The `@SerializedPersonalData` annotation has an optional `replacement` attribute which defaults to the empty string. This attribute is intended to control the replacement value, i.e. the value that the module will put in field if it is deleted (because it notices upon decryption that the required key is no longer available). On a `@SerializedPersonalData` field, this attribute is ignored by the default `ReplacementValueProvider` but it can be used by a custom version. See Section 4.4 for more details about this.

3.3.3. The `storageField` attribute

The `@SerializedPersonalData` annotation has an optional `storageField` attribute which defaults to the empty string. This can be used to control the actual field it uses for storage. If this attribute is empty, the module will look for the field name suffixed by 'Encrypted' as a storage field. This can be overridden by explicitly setting a storage field name with this annotation. For instance, the following might be used:

```
1 @SerializedPersonalData(storageField = "dobSecret")
2 LocalDate dateOfBirth;
3 byte[] dobSecret;
```

3.4. The `@DeepPersonalData` annotation

The `@DeepPersonalData` annotation is used for those cases where a field shouldn't be encrypted in its entirety, but should instead be recursively examined for GDPR Module annotations and processed accordingly. In this way, the GDPR Module can process more complex object graphs. An example use case would be this:

```
1 public class Person {
2     @DataSubjectId UUID id;
3     @PersonalData String name;
4     @DeepPersonalData Address address;
5 }
6
7 public class Address {
8     @PersonalData String line1;
9     @PersonalData String line2;
10    String country;
11 }
```

In this case, while processing a `Person`, the module would recurse into the `Address` object. As a result, the `Address.line1` and `Address.line2` fields would get encrypted and can be deleted. The `Address.country` field would remain clear. Note that we don't need another `@DataSubjectId` field in the `Address` class - this will be available from the context of processing the `Person` object. Of course, the `Address` class *could* have overridden the key id by having its own `@DataSubjectId` field.

The `@DeepPersonalData` annotation is allowed on fields with a type that carries at least one of the GDPR Module annotations (by itself or through a supertype), and any arrays/collections thereof (following the exact same recursion rules as described in Section 3.2).

3.4.1. The `scope` attribute

The `@DeepPersonalData` annotation has an optional `scope` attribute which functions exactly as for `@PersonalData`, described in Section 3.2.2.

3.5. The `@PersonalDataType` annotation

The `@PersonalDataType` annotation is a rarely-used class annotation (all other GDPR Module annotations are field annotations). It is used in those cases where we want to use `@DeepPersonalData` on a field of a particular type, in cases when that type as such doesn't carry any other GDPR Module annotations. The reason for still using `@DeepPersonalData` would be that we know that some or all of the subtypes do or may have such annotations. By the rules of `@DeepPersonalData`, this wouldn't be allowed. This can be fixed by annotating the supertype with the `@PersonalDataType` annotation.

For example, we could write the following to model that a `Person` has a set of relations; each `Relation` may be another `Person` or a `Pet`.

```
1 @PersonalDataType
2 public interface Relation {
3 }
4
5 public class Person implements Relation {
6     @DataSubjectId UUID id;
7     @DeepPersonalData Set<Relation> relations;
8 }
9
10 public class Pet implements Relation {
11     String name;
12 }
```

Without the `@PersonalDataType` annotation on line 1, the GDPR Module wouldn't accept the `@DeepPersonalData` on line 7, because the `Relation` type doesn't have any other GDPR Module field annotations.

3.6. Meta-annotations

All annotations in the GDPR Module can also be used as *meta-annotations*: you can define your own annotation types, put one or more GDPR Module annotations on the annotation definition, and then whenever you apply that annotation it is as if the GDPR Module annotation gets applied.

For instance, if you prefer the name `Erasable` over `PersonalData`, you might write:

```
1 @PersonalData
2 @Retention(RetentionPolicy.RUNTIME)
3 @Target(ElementType.FIELD)
4 public @interface Erasable {
5 }
```

and then apply this in your own code like:

```
1 public class NewCustomerEvent {
2     @DataSubjectId private int id;
3     @Erasable private String name;
4 }
```

3.7. Scala support

Although the annotations defined by the GDPR Module and described in the previous sections can in principle be used directly in Scala applications, they suffer from two disadvantages:

- When applied on a Scala class parameter (which is a very commons use case, since it's natural to model

events as Scala case classes), the annotation would by default only apply to the constructor parameter and not to the implied field. As a result, the annotation would not be detected by the `FieldEncrypter`. This can be fixed by applying `scala.annotation.meta.field`, but it leads to ugly code if that has to be applied every time.

- Conventionally, Scala prefers lower camel case annotation names rather than upper camel case.

To overcome these disadvantages, the module includes a Scala package object for `io.axoniq.gdpr.api` with lower camel case type aliases for all 5 annotations defined by the module. For the annotations to be used on fields (all except `@PersonalDataType`), these type aliases include the `@field` meta-annotation, so that this doesn't have to be included anymore in the application code.

Using these type aliases, Scala application code can look like this:

```
1 case class NewCustomerEvent(
2     @dataSubjectId id: Int,
3     @personalData name: String
4 )
```

Please note that Java annotations are completely separate from Scala annotations, and the GDPR Module only deals with Java annotations. The lower camel case versions of the annotations are Scala type aliases to Java annotations rather than Scala annotations. The module will *not* detect GDPR Module annotations attached as meta-annotations (see Section 3.6) to Scala annotations.

Chapter 4. Configuration and integration

4.1. The `CryptoEngine`

4.1.1. General

`CryptoEngine` is an interface that describes some capabilities that are at the core of the GDPR Module:

- Retrieving an existing key (returning nothing if it isn't there) - keys being represented as Java cryptography `SecretKey` objects.
- For a given id, either retrieving the existing key if it exists, or generating a new key, storing it and returning it if it doesn't.
- Deleting a key.
- Obtaining a Java cryptography `Cipher` object for performing actual encryption.
- Overriding the default length for new keys.

Normally, only the key deletion operation would be used directly by application code. The other operations are used by the `FieldEncrypter` class.

The reason for coupling the management of keys with the creation of `Cipher` objects is that this allows us to support a wide range of key management scenarios. One scenario is where keys are stored in a database and encryption is performed in the Java process itself. In this case, the two aspects could be fully separated. However, it is also possible to use a Hardware Security Module (HSM) in which case encryption has to take place on the device itself. In this case, a `SecretKey` object wouldn't actually hold any key material, but just a reference to the key on the HSM. This makes the key management and `Cipher` functionality closely related.

The GDPR Module has one abstract class implementing `CryptoEngine`, and 6 concrete classes. Their relation is shown in Figure 2.

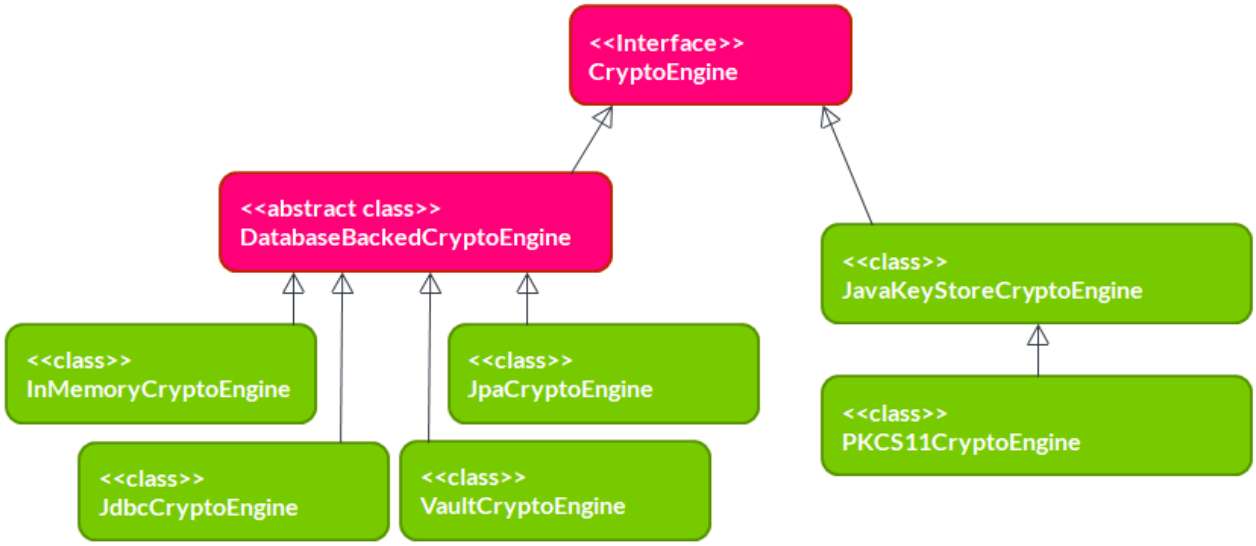

Figure 2. CryptoEngine class diagram

The 6 concrete implementations can be used as follows:

- `InMemoryCryptoEngine` keeps the keys in a local `HashMap`. This is very useful for writing unit tests, but not for using it with any production data.
- `JpaCryptoEngine` uses JPA to store keys in a relational database. As its constructor arguments, it accepts an `EntityManagerFactory` and optionally a JPA-mapped class used to store keys. This class must implement `KeyEntity`. If the class is not specified, it will use `DefaultKeyEntity`. Of course, the class provided to the constructor must be present in the persistence unit managed by the `EntityManagerFactory`. The `JpaCryptoEngine` will manage its own transactions so it doesn't need any reference to a transaction manager, and the persistence unit must have `RESOURCE_LOCAL` transaction type.
- `JdbcCryptoEngine` is like the `JpaCryptoEngine` in that it stores keys in a relational database, but it uses plain JDBC instead of JPA. It needs a `DataSource` object as a constructor parameter. It offers default table and column names, which can be overridden by optional constructor parameters.
- `VaultCryptoEngine` stores keys in HashiCorp Vault. The specifics of this implementation are described in Section 4.1.3.
- `JavaKeyStoreCryptoEngine` uses a Java cryptography `KeyStore` implementation to store its keys. It will examine the given `KeyStore` for its provider, and then obtain a `KeyGenerator` and `Cipher` object from this provider as well. Please note the default Java `KeyStore` implementation used for instance to store certificates for a web server, is totally unsuitable for this particular application. The main reason why this class exists is as a stepping stone to implement the PKCS#11 implementation.
- `PKCS11CryptoEngine` supports HSMs using the PKCS#11 standard through the SunPKCS11 JCE provider. It needs a PKCS#11 configuration (filename or stream) and a password as constructor arguments. It will then try to instantiate the PKCS#11 provider, and open the keystore using the password. (And if successful, it will delete the password for security reasons.)

4.1.2. Safety of concurrent access

When reading keys, `CryptoEngine` safety of concurrent access is trivial. In the case of generating a new key, there is a real risk: two parallel processes may both find a key to be absent, generate one, save it, and then start encrypting fields with it. The second key to be saved would overwrite the first key, causing data loss because data encrypted with the first key will now no longer be decryptable. It will vary from application to application whether this is a realistic scenario.

Looking at the implementations provided:

`InMemoryCryptoEngine`

: Only operates in a single process, reducing the general concurrency problem to a simpler thread-safety problem. The `InMemoryCryptoEngine` implements this safety by using the atomic operations of `ConcurrentHashMap` in its implementation.

`JpaCryptoEngine`

`JdbcCryptoEngine`

: These implementations are designed to be used on multiple nodes, pointing to the same database. They are fully safe for concurrent access, and use database transactions to implement this.

`VaultCryptoEngine`

: This implementation is designed to be used on multiple nodes, pointing to the same Vault server/cluster. This is fully safe for concurrent access.

`JavaKeyStoreCryptoEngine`

`PKCS11CryptoEngine`

: These implementations are ***not*** safe for concurrent access. If this is a realistic scenario for your application, additional safeguards are necessary.

4.1.3. The `VaultCryptoEngine`

The `VaultCryptoEngine` uses HashiCorp Vault to store keys. This has a number of interesting benefits: Vault offers encryption of keys, protected by a master key which isn't stored anywhere. The master key can be provided manually on startup, but can (in the enterprise version) also be obtained through a HSM, through AWS KSM or through Google KSM. The individual keys can be stored in a wide range of configurable backends. Using HashiCorp Vault, you can achieve a high level of key security while still maintaining high performance.

The `VaultCryptoEngineTest` provides a complete example of how to instantiate this engine.

Vault Client

HashiCorp Vault has an HTTP API. There is no official Java client library available. (The only official ones are for

Go and Ruby.)

The GDPR Module doesn't use a community Java library, but instead uses OkHttp as a fast, lower-level HTTP client. Therefore, the module has an optional dependency on OkHttp, which must be present for using the `VaultCryptoEngine`. The constructor of `VaultCryptoEngine` requires an `OkHttpClient` object. OkHttp is performance-optimized, and a single `OkHttpClient` can be used across all threads. This makes it a great fit for the GDPR Module use case.

In the *test* code we also use the community BetterCloud vault client to set a Vault policy and create a token. This client is easy to use and has a wide range of functionality, but isn't suitable as the overall client for the GDPR Module because of performance limitations when using SSL.

Policy

An essential operation for the GDPR Module is to store a new key if and only if no key is stored yet for the key identifier. On a relational database, this can be implemented easily by create a unique constraint for the key identifier: an INSERT on an existing identifier will then fail. On Vault, this is not possible. POST and PUT calls are treated exactly the same, and in both cases will result in either a create or an update, depending on whether the entry was already there. This does not fit well with the GDPR Module. The way to solve this is to properly using Vault authorizations.

The `VaultCryptoEngine` **MUST** operate under a policy that allows 'create', 'read' and 'delete' but doesn't allow 'update'. This will prevent undesired overwrites of keys.

The `VaultCryptoEngineTest` uploads such as policy to Vault each time it runs.

Token

When making calls to Vault, the GDPR Module will authenticate by providing a *token* in the `X-Vault-Token` HTTP header. It is up to the application to obtain such a token and provide it to the `VaultCryptoEngine`. Vault supports a wide range of authentication methods to obtain a token.

A token must be provided when constructing a `VaultCryptoEngine`, but may also be updated at any point using the `setToken` method. Since tokens have limited timespan, applications may need to call this method periodically to refresh the token.

The `VaultCryptoEngine` does not have any built-in mechanism to refresh the token or detect token staleness. The application must supply a new token to the `VaultCryptoEngine` in a timely manner.

The `VaultCryptoEngineTest` uses the BetterCloud Vault client and a root token to obtain the client token for the GDPR Module itself.

4.2. The `FieldEncrypter`

The `FieldEncrypter` class can encrypt and decrypt classes having GDPR Module annotations. Other objects may be processed by it, it will simply leave those unaltered. This class may be used directly by applications, but in most Axon Framework-based applications it would be used by the `FieldEncryptingSerializer` instead of being used directly. In addition to having encrypt/decrypt operations, it has a `willProcess` method to determine whether or not a particular object would potentially be modified by the encrypt/decrypt operation. This allows for some performance optimalizations in the `FieldEncryptingSerializer`.

An important property realized by the `FieldEncrypter` is *idempotency*: encrypting an already-encrypted field will simply leave the field untouched, and similarly decrypting an non-encrypted field will leave the field in its current state. The `FieldEncrypter` is also `null`-safe: `null` values will simply remain `null` upon encryption and decryption.

The `FieldEncrypter` constructor needs as a minimum a `CryptoEngine` implementation. Additionally, you may provide it with a `Serializer` to be used for processing `@SerializedPersonalData` fields. (If not present, it will use an `XStreamSerializer` by default.) You may also provide it with a `ReplacementValueProvider` - if not present, it will use the GDPR Module's default implementation.

The `FieldEncrypter` has a `setModifyImmutableCollections(boolean value)` method which determines how it processed immutable Java collections; see Section 3.2.

4.2.1. Using preset encryption contexts

When encrypting and decrypting fields using the `FieldEncrypter`, key identifiers are normally obtained by inspecting the object's `@DataSubjectId` fields. In some cases, this may not work because there is no field in the object that contains an appropriate key id. This is in fact common in use cases where the module is used to protect confidential data from accidental disclosure rather than enabling cryptographic erasure. In those cases, it may make sense to use a system wide key.

These use cases are supported by the GDPR Module through additional methods on `FieldEncrypter` beyond the default `#encrypt(Object)` and `#decrypt(Object)`. The additional methods allow the application to specify one or more preset keys that will always be there regardless of `@DataSubjectId` annotations. Still, these keys are treated as a default only and may be overridden by subsequent `@DataSubjectId` fields, much in the same way a objects deeper in the object graph may override a keyId set by an object higher in the graph.

Specifically, the methods signatures are the following:

```
encrypt(Object object, String keyId);
decrypt(Object object, String keyId);
encrypt(Object object, Map<String, String> keyIds);
decrypt(Object object, Map<String, String> keyIds);
```

When only providing a `String` keyId, this keyId will be included in the context with the default, empty group. When providing a `Map`, the elements will be treated as group/keyId pairs.

4.2.2. Performing partial encryption/decryption

Normally, the `FieldEncrypter` will process and act on all GDPR Module annotations. In some cases, it may be desirable to exercise more control over this process. The `FieldEncrypter` does offer the possibility of specifying a `Set` of groups (which are included in `@DataSubjectId`, `@PersonalData` and `@SerializedPersonalData` which must be processed. If this set is `null` (which is the default), all groups will be processed.

There are two ways to specify this set of groups:

1. By invoking the `setGroups` method of the `FieldEncrypter`. After doing this, the specified set of groups becomes the default for all subsequent `encrypt` and `decrypt` operations. This can also be used to limit the groups in conjunction with a `FieldEncryptingSerializer`.
2. By specifying the set of groups as an argument to `encrypt` or `decrypt`. In this case, it only applies to that particular invocation. The value provided in this way overrides any value set through the first method.

4.2.3. Performing immediate replacement

In some cases, it is useful to immediately replace any cleartext value with the value that it would effectively have after crypto-erasure. This could be achieved by encrypting, then deleting the key, and then do decrypting. However, this is cumbersome and a waste of resources in this scenario. Therefore, there is a specific `replace` call in the API that does this immediately while avoiding all encryption.

The `replace` call supports the partial encryption/decryption mechanism described in Section 4.2.2.

The `replace` call also supports the custom replacement value mechanism described in Section 4.4.

Specifically, the `FieldEncrypter` does the following when processing a `replace` call:

- When processing a field, it detects whether it is encrypted or not. If it's encrypted, it obtains the storedPartialValue from the encrypted data. (None of that is a cryptographic operation by itself.) If it's not encrypted, it obtains a storedPartialValue from invoking `ReplacementValueProvider#partialValueForStorage` on the cleartext.

- Then, it directly invokes `ReplacementValueProvider#replacementValue` to obtain a new value.

4.3. The `FieldEncryptingSerializer`

The `FieldEncryptingSerializer` is an implementation of `Serializer` (either Axon 3, Axon 2 or the embedded one, depending on your version). It takes another `Serializer` as a constructor argument and wraps this one. When serializing, it will perform encryption first using an encapsulated `FieldEncrypter`, and then perform serialization. It will do the reverse upon deserialization. This allows for very elegant integration into an Axon Framework application.

Assuming Axon 3 on Spring and a JPA `CryptoEngine`, configuration of the GDPR Module could be as easy as this:

```
1 @Bean
2 public CryptoEngine cryptoEngine(EntityManagerFactory emf) {
3     return new JpaCryptoEngine(emf);
4 }
5
6 @Bean
7 public Serializer serializer(CryptoEngine cryptoEngine) {
8     return new FieldEncryptingSerializer(cryptoEngine,
9         new XStreamSerializer());
10 }
```

This would be sufficient to make sure that encryption takes place whenever events get serialized, which means they will always end up in their encrypted form in an event store.

The constructor may be provided with a `FieldEncrypter` for detailed control. If the constructor gets a `CryptoEngine` and optionally a `ReplacementValueProvider`, it will construct it `FieldEncrypter` itself where the `Serializer` used by the `FieldEncrypter` for dealing with `@SerializedPersonalData` is the same one as the delegate serializer of the `FieldEncryptingSerializer` (which would be a reasonable choice in many cases).

4.4. The `ReplacementValueProvider`

The `ReplacementValueProvider` class is used to determine the behaviour of the GDPR Module in case it tries to decrypt a field for which a key has been deleted. The default `ReplacementValueProvider` will lead to the following behaviour: On any `String` field, it will replace that field value by the value of the `replacement` attribute of the `@PersonalData` annotation. In other cases, the field value will be `null` (except for primitives, which will get the Java default value).

By subclassing `ReplacementValueProvider` and providing this implementation to the `FieldEncrypter`, this behaviour can be modified. There are two methods that be overridden:

- `replacementValue` gets invoked when decryption can't take place. The return value of this method will be put in the field. The method gets a large number of arguments which it can use to determine the appropriate value of the field - please see the Javadoc for a complete list.
- `partialValueForStorage` gets invoked when encrypting a value. The default implementation returns `null`. It may return a piece of clear text data (as a `byte[]`), which will be stored alongside the encrypted data. This data will be available for the `replacementValue` method upon failed decryption. This functionality can be used to do things like deleting a full date but keeping the year field, or deleting a full credit card number but keeping the last 4 digits.

For instance, suppose you store a date of birth in `LocalDate` field. The full date of birth may be considered personal data which must be erased, whereas the year or it by itself would not be personal information and may still be kept for analytics purposes. This could be done by setting the `LocalDate` to a year with a fixed month/day of January 1st. To indicate to the GDPR Module to proceed like this, we might want to use the constant **`YEARONLY`** as value of the replacement attribute, like this:

```
1 @SerializedPersonalData(replacement = "*YEARONLY*")
2 LocalDate dateOfBirth;
3 byte[] dateOfBirthEncrypted;
```

With the default `ReplacementValueProvider`, this won't achieve anything, we we may easily subclass the `ReplacementValueProvider` to implement this desired behaviour, like this:

```
public static class YearOnlyReplacementValueProvider
        extends ReplacementValueProvider {
    @Override
    public Object replacementValue(Class<?> clazz, Field field,
            Type fieldType, String groupName, String replacement,
            byte[] storedPartialValue) {
        if(LocalDate.class.equals(fieldType) &&
                "*YEARONLY*".equals(replacement) &&
                storedPartialValue != null) {
            ByteBuffer buffer = ByteBuffer.allocate(Integer.BYTES);
            buffer.put(storedPartialValue);
            buffer.flip();
            return LocalDate.of(buffer.getInt(), Month.JANUARY, 1);
        } else {
            return super.replacementValue(clazz, field,
                fieldType, groupName, replacement, storedPartialValue);
        }
    }

    @Override
    public byte[] partialValueForStorage(Class<?> clazz, Field field,
            Type fieldType, String groupName, String replacement,
            Object inputValue) {
        if(LocalDate.class.equals(fieldType) &&
                "*YEARONLY*".equals(replacement) &&
                inputValue instanceof LocalDate) {
            ByteBuffer buffer = ByteBuffer.allocate(Integer.BYTES);
            buffer.putInt(((LocalDate)inputValue).getYear());
            return buffer.array();
        } else {
            return super.partialValueForStorage(clazz, field,
                fieldType, groupName, replacement, inputValue);
        }
    }
}
```

Upon encryption, the `partialValueForStorage` method would extract the `year` from the `LocalDate` and store this as a sequence of bytes. Upon a failed decryption attempt, the GDPR Module would invoke the `replacementValue` method where these stored bytes would be available as the `storedPartialValue` parameter. Based on that, it will return a `LocalDate` instance with month and day set to January 1st regardless of the original value.

4.5. Exception handling

The GDPR Module defines two exceptions, both of them being `RuntimeException`. These are:

`ConfigurationException`

> This is used to report situations where something is wrong with the way the system has been configured, which may be in the setup of the `CryptoEngine`, or in the placement of annotations on data classes.

`DataException`

This is used to report situation where something is wrong with an indvidual data record, but the system as a whole can continue.

In general, it may make sense to catch `DataException` to attempt some recovery. A `ConfigurationException` would usually be indicative that something is wrong with the code and/or the platform and is therefore unrecoverable within the platform.

Exceptions thrown by the GDPR Module have an error code. Error codes have the following ranges:

GDPR-0000

internal exceptions (should never occur - indicates a bug in the GDPR Module)

GDPR-1000

platform exceptions (should never occur - indicates that the Java platform doesn't meet all standards)

GDPR-2000

platform warnings, in particular unability to reflectively modify unmodifiable collections

GDPR-3000

license exceptions

GDPR-4000

configuration exceptions

GDPR-5000

keystore-related exceptions (which may be `ConfigurationException` or `DataException`)

GDPR-6000

data-related exceptions

Chapter 5. Technical background

5.1. Introduction

This chapter describes all details of the encryption procedure used by the GDPR Module. This knowledge is not required for using the GDPR Module. We provide it nevertheless, for the sake of transparency. We expect that some organization will want to evaluate this as a way to get assurance on the security provided by the GDPR Module. Also, it serves as a protection against vendor lock-in: using the information provided in this chapter, it will always be possible to decrypt data encrypted by the GDPR Module (assuming the keys are still there), without actually using the GDPR Module itself.

5.2. Encryption algorithm and key sizes

Encryption is at the core of the GDPR Module, so it's crucial that to choose a strong and appropriate encryption algorithm. The fact that the GDPR Module relies on the long-term security of the algorithm makes this need even greater.

Encryption algorithms can broadly be divided into *asymmetric* algorithms, using a public/private keypair (prime examples include RSA and Elliptic Curve Cryptography (ECC)), and *symmetric* algorithms, using a single secret key (examples include Triple DES, Blowfish and AES). Given the nature of its operations, the GDPR Module needs a symmetric algorithm. This also gives some great benefits: compared to asymmetric algorithms, symmetric algorithms are a lot faster, and require smaller key sizes.

The symmetric encryption algorithm chosen is the industry-standard Advanced Encryption Standard (AES) - the same algorithm used to protect top-secret military information.

AES can operate at three key sizes: 128, 192 and 256 bits. While all key sizes provide adequate security today and for years to come, the security landscape may change in the somewhat more distant future once quantum computing arrives. In this post-quantum world, some present-day cryptographic algorithms will no longer provide any security at all (importantly, this holds for RSA and ECC). The AES algorithm used by the GDPR Module will survive, but its security depends on the key length. AES-128 will likely be insecure in the post-quantum world whereas AES-256 would still be secure. (For more info, see for instance the recommendations from PQCRYPTO, an EU-funded research project into post-quantum cryptography for long-term security, http://www.pqcrypto.eu.org/docs/initial-recommendations.pdf.) For this reason, the GDPR Module defaults to using AES-256. This can be overridden to AES-192 or AES-128, but we recommend to leave it at the default.

5.3. Encryption modes

AES is a block cipher, encrypting individual blocks of 128 bits (16 bytes). Actual data to be encrypted is usually not exactly 16 bytes. This means that we need to choose a way to apply the algorithm to a larger piece of data

(called the *mode*), and we need to decide on a way to deal with data that is less than 16 bytes or not an even multiple of 16 bytes (called the *padding*).

The simplest *mode* is to simply chunk up the clear data and encrypt it block by block. This is called Electronic Code Book (ECB) mode. This is generally insecure, because by looking at the cryptotext you could still identify which blocks are equal to others, which may give a lot of information. (This is especially obvious when applying this to some image formats.)

For this reason, the GDPR Module uses a different mode, called Cipher Block Chaining (CBC). With CBC, the encrypted data from one block is XOR'ed with the cleartext data of the next block before encrypting that block. This overcomes the weakness of ECB.

This leaves the question of what to to with the first block, since there is no prior block of encrypted data to use. Here, we use a preset value called the Initialization Vector (IV). We could use a constant (or maybe all-zero) IV, but this would again introduce a security issue: the same field values encrypted by the same key, would yield the same encrypted data, which may provide information to an attacker: so-called *semantic security* is not guaranteed. Therefore, the GDPR Module does use an IV. A full AES-IV is 16 bytes. To save on some space, we use a 4-byte *pre_iv* value to derive a 16-byte IV - which still provides a more than sufficient level of semantic security.

For padding, we use the industry-standard PKCS#5 algorithm.

5.4. Auxiliary cryptographic algorithms

While AES in CBC mode with PKCS#5 padding is at the heart of our cryptosystem, some other auxiliary algorithms will be used as well.

The MD5 hash function is used to generate digests (for detecting GDPR Module encryption) and to generate the full IV from the pre_iv. MD5 is by itself no longer considered a *secure* hash function (in the sense of being collision resistant), but that's totally fine for the GDPR Module application of it: it doesn't rely on MD5 security at all. MD5 happens to be a convenient choice because it is fast and because the MD5 digest size equals the AES block (and therefore IV) size.

In addition to using AES in CBC mode with PKCS#5, the system also uses AES in ECB mode without any padding. This is used to encrypt a single MD5 digest into an encrypted MD5 digest. In this application, ECB mode is secure.

5.5. Auxiliary encoding algorithms

To convert a Java `String` into a `byte[]`, UTF-8 decoding will be used. To store a `byte[]` value in a `String` field, Base64 encoding will be used.

When we're converting between `byte[]` and Java primitive types `int` and `long`, we use big-endian encoding.

To store more complex data structures into a single `byte[]`, the module uses the Protocol Buffers protocol (see https://developers.google.com/protocol-buffers/). The associated `.proto` file is included in the software distribution.

5.6. Encryption of byte arrays

Encrypting a `byte[]` field is the most basic encryption case of the system. The `String` and general serialized object cases use the process for `byte[]` encryption as a building block. The process is illustrated in Figure 3.

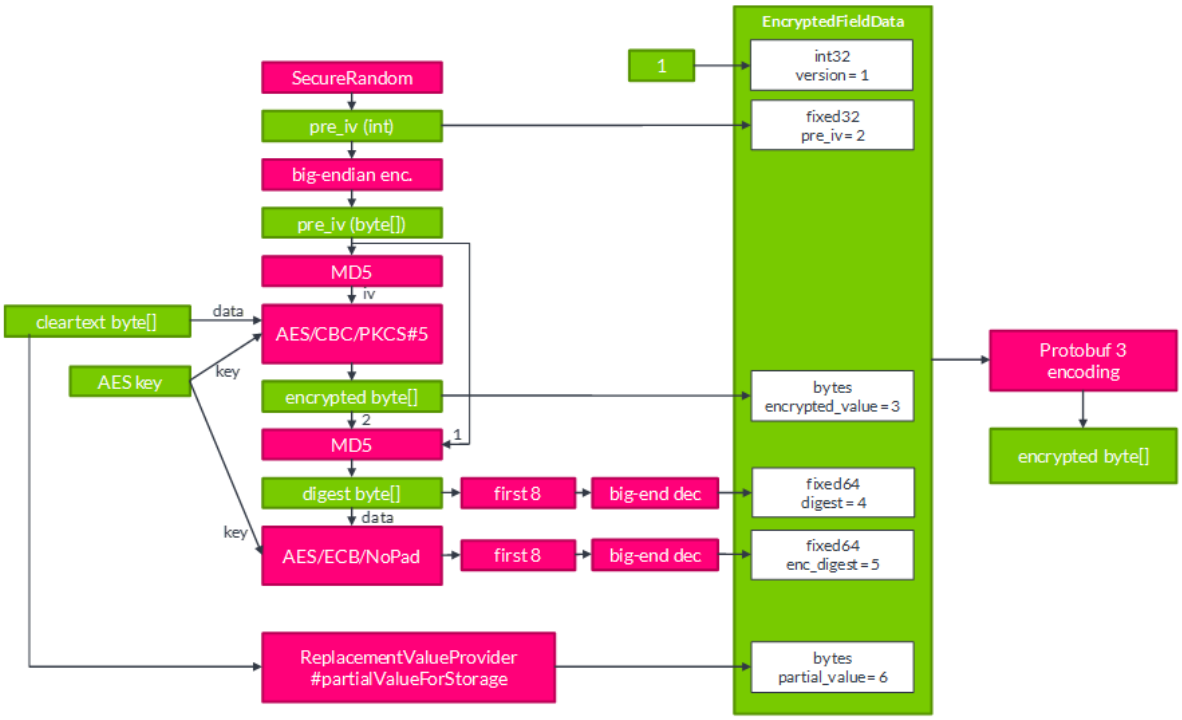

Figure 3. Byte array encryption process

Input to the process is a cleartext `byte[]` and an AES key. We'll generate a random `int` as pre_iv from which we derive an actual IV through MD5. This gives us everything to perform AES/CBC/PKCS#5.

Since we'll store the `byte[]` data in another `byte[]` field, the module must be able to reliably distinguish between a `byte[]` value which is clear and a `byte[]` value which is encrypted - there is no other `flag` field to keep that information. To do this, the module will also store a 64-bit digest of the pre_iv combined with the encrypted data. This digest is created by using MD5, and then interpreting the first 8 bytes of the digest as a `long`.

Another case to deal with is this: if a key for a certain id is created, data gets encrypted with that key, the key gets deleted, and then a new key is recreated for that same id, the module might try to decrypt data with the wrong key. There is no general way of dealing with this in AES, and the result may either be an exception or simply wrong data (dependent on the padding). Of course, it is desirable that this will simply be treated as a missing key case. To achieve this, we'll also store an encrypted version of the digest. This is created by taking the full MD5 digest, applying AES/ECB (no padding required), and again interpreting the first 8 bytes as a long.

Upon decryption, the module repeats this procedure prior to attempting decryption, to ensure that it has the correct key.

Secondary to the encryption process, the module will try to obtain a partial, clear `byte[]` value from the `ReplacementValueProvider` (see Section 4.4).

Now, all components are in place to generate the final `byte[]` value. The fields of a Protocol Buffer defined `EncryptedFieldData` object will be filled with the data obtained so far. There is a `version` field which is always set to `1` in the current version - this may help us to implement new features in future versions. This object will be transformed into a `byte[]` using the Protocol Buffer algorithm.

5.7. Encryption of strings

Encryption of a `String` is illustrated in Figure 4. The `String` is transformed into `byte[]` with UTF-8 before the byte array encryption process is applied. As input for the `ReplacementValueProvider`, the original `String` will be used rather than the `byte[]`. To get to a `String` again, the output from the byte array encryption process will be transformed using Base64.

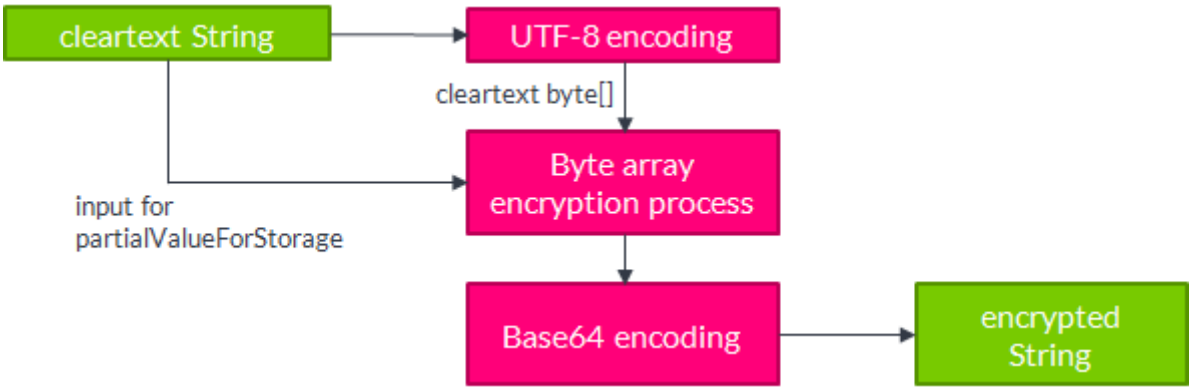

Figure 4. String encryption process

5.8. Encryption of other objects

Encryption of types other than `byte[]` and `String` requires serialization as illustrated in Figure 5. The exact serialization class differs between the axon3, axon2 and core versions of the GDPR Module, but it follows the same logic. The serializer is designed to serialize into a number of different types, but within the GDPR Module we'll always serialize to the content type `byte[]`. The result of serialization is a `SerializedObject` consisting of the content type (always `byte[]`), the type name and revision, and the actual data. The data will be encrypted using the byte array encryption process. The original object (rather than its serialized form) is input to the `ReplacementValueProvider`.

To transform the combined results into `byte[]`, we'll store the data in a Protocol Buffer defined `SerializedEncryptedFieldData` object (along with a fixed `version` of `1`) and apply Protocol Buffer serialization. In case the storage field is `byte[]` type, we're done. In case the storage field is `String` type, we'll

do a final conversion using Base64.

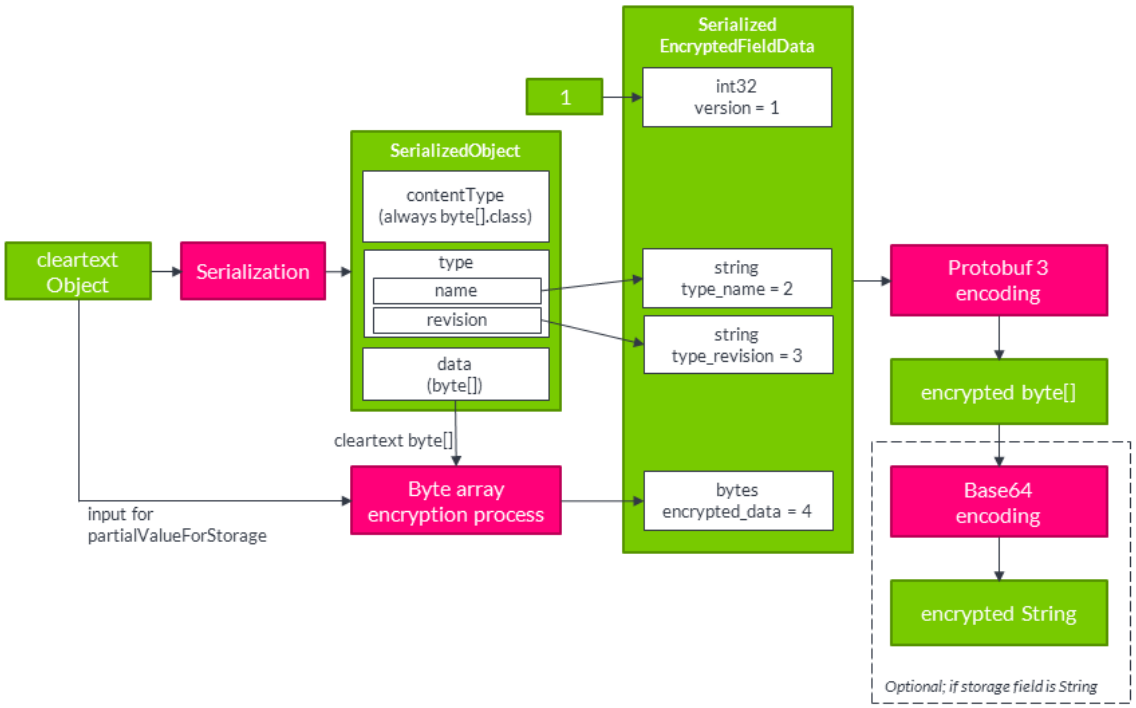

Figure 5. Serialized object encryption process

Appendix A: Change log

1.2

- Added functionality for immediate replacement (see Section 4.2.3).
- Added support for HashiCorp Vault (see Section 4.1.3).
- Enabled placing the license file on the classpath (see Section 2.1).

1.1

- Added support for Scala, including type aliasas for the GDPR Module annotations (see Section 3.7) and support for Scala collections (see Section 3.2).
- Added a JDBC-based CryptoEngine (see Section 4.1).
- Added list of included licenses in this guide, in Appendix B.

1.0.5

- Added functionality for partial and double encryption. See Section 3.2.4 and Section 4.2.2.

1.0.4

- Added functionality to include a preset encryption context in `FieldEncrypter#encrypt` and `#decrypt`, to support use cases where there is no field that can serve as `@DataSubjectId`. (See Section 4.2.1.)

1.0.3

- Corrected handling of `null` values with `@SerializedPersonalData`.
- Changed `license` system property into `axoniq.gdpr.license` to avoid potential collisions.
- Changed `axon-core` Maven dependency into optional to avoid the GDPR Module from overriding the project Axon Framework version.

1.0.2

- Corrected typo in manual in Figure 1.

1.0.1

- Corrected handling of primitive types.

Appendix B: Included libraries and licenses

Library	License
Jackson libraries by FasterXML https://github.com/FasterXML	Apache 2.0 http://www.apache.org/licenses/LICENSE-2.0
Google Gson https://github.com/google/gson	Apache 2.0 http://www.apache.org/licenses/LICENSE-2.0
Google Guava https://github.com/google/guava	Apache 2.0 http://www.apache.org/licenses/LICENSE-2.0
Google Protobuf for Java https://github.com/google/protobuf	BSD 3-Clause license, https://opensource.org/licenses/BSD-3-Clause
Googlecode Gentyref https://code.google.com/archive/p/gentyref/	Apache 2.0 http://www.apache.org/licenses/LICENSE-2.0
DOM4J https://dom4j.github.io/	Custom, BSD-like: https://github.com/dom4j/dom4j/blob/master/LICENSE
Hibernate JPA API https://github.com/hibernate/hibernate-jpa-api	EDL https://www.eclipse.org/org/documents/edl-v10.php
OkHttp http://square.github.io/okhttp/	Apache 2.0 http://www.apache.org/licenses/LICENSE-2.0
SLF4J API https://www.slf4j.org/	MIT https://www.slf4j.org/license.html
Xalan https://xml.apache.org/xalan-j/	Apache 1.0 http://www.apache.org/licenses/LICENSE-1.0
Xerces http://xerces.apache.org/	Apache 2.0 http://www.apache.org/licenses/LICENSE-2.0
xmlpull http://www.xmlpull.org/	public domain
XOM https://github.com/Malax/XOM	LPGL-2.1 https://www.gnu.org/licenses/old-licenses/lgpl-2.1.en.html
XPP3 https://www.extreme.indiana.edu/xgws/xsoap/xpp/	Apache 1.0 http://www.apache.org/licenses/LICENSE-1.0