broadway-sensitive-serializer

Release 0.1

Matteo Galacci

Sep 19, 2022

CONTENTS

The idea behind this project is to make a **CQRS+ES** system compliant, specifically implemented through the Broadway library, with the general data protection regulation (GDPR), *in particular with the right to be forgotten*.

CHAPTER

ONE

CONTENTS

—

1.1 Basic concepts

1.1.1 CQRS

CQRS (Command Query Responsibility Segregation) is a pattern that aims to separate responsibilities for queries and for commands. It is a pattern that separates the read and write operations on a given model. This, in practice, leads to different concrete objects, separated in write models and read models. So, this pattern can lead to different tables or data stores where the data is separated based on whether it is a command (write) or a query (read). Separation apart, the last state of the model will still be persisted as it happens in traditional CRUD systems.

1.1.2 Event Sourcing

ES (Event Sourcing), used together with CQRS, "transforms" the writing part of the CQRS models into a succession of events that are persisted in an Event Store, a specific table or data store that acts as a chronological and immutable register of events. The idea is that commands executed on a model lead to the issue of events which are stored in the Event Store. In this table are persisted all events issued by a Model, with a specific incremental index, sometimes called *playhead*, that represents order in which the events have been issued; for each new Model (Aggregate), its events starts with *playhead = 0*. Event Store is therefore recording system for all events that, if re-applied to the model in same order of generation, bring it to its last state. Or it might be possible to see a previous status of a model. These events then, if listened to specific Listeners, can project views (Read Models) or generate new commands (Processor). The views will then be the models (persisted in tables other than the Event Store or even a different Data Store) used by the read queries.

1.1.3 Event Store immutability

So using whole CQRS+ES pattern, we have an Event Store in which all events will be written in chronological order and grouped for each model using aggregate id. Event Store is immutable by its nature; after writing an event, it can never change. If necessary, compensation events will be issued to compensate the previous events. Imagine a bank account and its list of transaction, and think of a compensation event as a reversal.

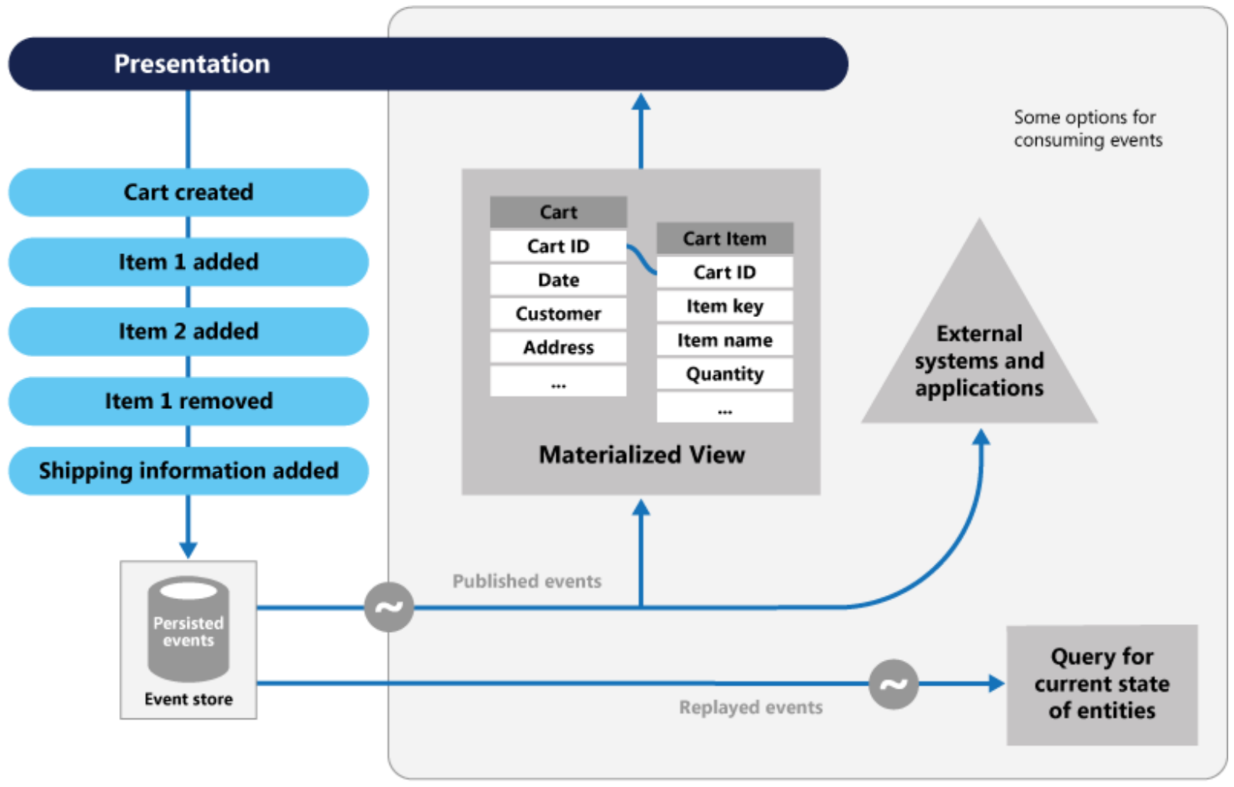

Fig. 1: CQRS+ES diagram from Microsoft's website

1.1.4 Projections

In a CQRS+ES system there are usually projections. If the event store is the chronological register of all the writing operations that took place on a specific Aggregate, then a projection is a specific view of the data; for a single Aggregate we could have as many views as there are our needs. So, after an event is issued, an event listener could listen that event in order to project a view of it. Multiple event listeners can listen same event to project different representations of the same data set.

1.1.5 Art. 17 GDPR -Right to be forgotten

The lay says: The data subject shall have the right to obtain from the controller the erasure of personal data concerning him or her without undue delay and the controller shall have the obligation to erase personal data without undue delay… Read the complete legislation

1.1.6 CQRS+ES+GDPR

We have said that in CQRS+ES pattern the Event Store is immutable and we have also said that to be compliant with the GDPR, a user can be request cancellation of his data. Thus said it seems a paradox, right? Because deleting user's data in a CQRS+ES system would mean either deleting events from Event Store or modifying existing events. Both things we cannot do. Compensation events cannot useful in this case as by going back in history, we could always recover user's data.

1.2 The proposal

This library proposes a solution to the problem about *CQRS+ES+GDPR*.

1.2.1 Event Store

Instead of thinking in terms of deleting or modifying events, the idea is to persist from the very beginning of the history, events in which payload (user information, or in general the event containing sensitive data) is encrypted by an encryption key specific for each Aggregate. As long as the key is present, the data can be encrypted and decrypted. When the key will be deleted (following a user request), the events will remain in the Event Store, but the payload, originally encrypted, will remain encrypted without the possibility of decryption. Thus, the story will remain unchanged, but the data is not understandable.

Normal payload

Sensitized payload

```
{
    "class": "SensitiveUser\\User\\Domain\\Event\\UserRegistered",
    "payload": {
        "id": "b0fce205-d816-46ac-886f-06de19236750",
        "name": "Matteo",
        "surname": "Galacci",
        "email": "m.galacci@gmail.com"
        "occurred_at": "2022-01-08T14:22:38.065+00:00",
    }
}
```

```
{
    "class": "SensitiveUser\\User\\Domain\\Event\\UserRegistered",
    "payload": {
        "id": "b0fce205-d816-46ac-886f-06de19236750",
        "name": "Matteo",
        "surname": "#-#2Iuofg4NKKPLAG2kdJrbmQ==:bxQo+zXfjUgrD0jHuht0mQ==",
        "email": "#-
↪#OFLfN9XDKtWrmCmUb6mhY0Iz2V6wtam0pcqs6vDJFRU=:bxQo+zXfjUgrD0jHuht0mQ==",
        "occurred_at": "2022-01-08T14:22:38.065+00:00",
    }
}
```

1.2.2 Projections

The sensitization operation is performed at a different time from the event projection, so the views will have the data decrypted to allow the read operations to work correctly. When a user makes use of the right to be forgotten, you should do three things:

1. Delete his encryption key
2. Delete the views that contain his data
3. Re-project events to regenerate views with encrypted data. (This will be easy as since there is no encryption key for a specific Aggregate, reading it from the Event Store will be hydrated with the sensitized data. This obviously involves *particular checks* in the Value Objects or in the Aggregate itself)

Fig. 2: Broadway sensitive serializer event store representation

1.2.3 Important note

It's important understand that the idea behind this project is not about general security or data leak. The idea behind this implementation is rather to make a CQRS + ES system compliant with the user's right of asking at any time to be forgotten, while keeping the system consistent.

Of course, you can use this library also in a different context of GDPR law, since that basically this library does nothing but decorate Broadway serializer, giving it the ability to encrypt and decrypt payload of the events.

1.3 Broadway concepts and proposal

1.3.1 Aggregate and persistence

Since this wiki is not meant to be a complete manual on the concepts in question, we will just call the Model, Aggregate and remind ourselves that the Aggregate, as such, is the source of our domain events; a client will ask the `User` Aggregate to create a new user, which will not only create the instance, but also the related event, `UserCreated`.

```
class BroadwayUsers extends EventSourcingRepository implements Users
{
    public function add(User $user): void
    {
        parent::save($user);
```

(continues on next page)

(continued from previous page)

```
    }
}

class User extends EventSourcedAggregateRoot
{
    public static function crea(
            UserId $userId,
            string $name,
            string $surname,
            string $email,
            DateTimeImmutable $regDate
    ): self
    {
        $user = new self();

        $user->apply(new UserCreated($userId, $name, $surname, $email, $regDate));

        return $user;
    }
}

$user = User::create($userId, $name, $surname, $email, $registrationDate);

$users->add($user);
```

1.3.2 Event serialization

When we ask Broadway to persist an Aggregate, the EventSourcingRepository takes all the events not yet committed from the Aggregate and asks the specific implementation of the Event Store to serialize them and then save them. For example, in the case of Broadway DBALEventStore:

```
private function insertMessage(Connection $connection, DomainMessage $domainMessage):␣
↪void
{
    $data = [
        'uuid' => $this->convertIdentifierToStorageValue((string) $domainMessage->
↪getId()),
        'playhead' => $domainMessage->getPlayhead(),
        'metadata' => json_encode($this->metadataSerializer->serialize($domainMessage-
↪>getMetadata())),
        'payload' => json_encode($this->payloadSerializer->serialize($domainMessage->
↪getPayload())),    // <-----
        'recorded_on' => $domainMessage->getRecordedOn()->toString(),
        'type' => $domainMessage->getType(),
    ];

    $connection->insert($this->tableName, $data);
}
```

When instantiating the `EventSourcingRepository` you need to inject a serializer (`$this->payloadSerializer`) which in the case of the default Broadway implementation is a `SimpleInterfaceSerializer` which implements the `Broadway\Serializer` interface. `SimpleInterfaceSerializer` does nothing but call the `Broadway\Serializer::serialize($object): array` or `Broadway\Serializer::deserialize(array $serializedObject)` method on the event to be serialized in the case of reading from the Event Store, where it is necessary to recreate the event starting from the payload.

Let's focus for now on the `Broadway\Serializer::serialize($object): array` method which, as read from the signature, returns an array which is later converted to json thanks to PHP's `json_encode()` function.

1.3.3 Proposal implementation

It is precisely on the serializer that this library intervenes. The idea is to decorate the native Broadway serialization by adding the ability to encrypt and decrypt (sensitize and desensitize) the payloads of the events, or rather the values of its keys, based on 3 strategies that we will see later, `Whole strategy`, `Partial Strategy` and `Custom strategy`. Therefore, when a new aggregate is created, a specific key will be generated which will be used to encrypt and decrypt.

DBAL payload

Whole strategy

Partial strategy

Custom strategy

```
{
    "class": "SensitiveUser\\User\\Domain\\Event\\UserRegistered",
    "payload": {
        "id": "446effc9-4f5c-4369-8e89-91cb5c8509b9",
        "occurred_at": "2022-01-08T14:22:38.065+00:00",
        "name": "Matteo",
        "surname": "Galacci",
        "email": "m.galacci@gmail.com"
    }
}
```

```
{
    "class": "SensitiveUser\\User\\Domain\\Event\\UserRegistered",
    "payload": {
        "email": "#-
↪#OFLfN9XDKtWrmCmUb6mhY0Iz2V6wtam0pcqs6vDJFRU=:bxQo+zXfjUgrD0jHuht0mQ==",
        "id": "b0fce205-d816-46ac-886f-06de19236750",
        "name": "#-#EXWLg\/JANMK\/M+DmlpnOyQ==:bxQo+zXfjUgrD0jHuht0mQ==",
        "occurred_at": "2022-01-08T14:25:13.483+00:00",
        "surname": "#-#2Iuofg4NKKPLAG2kdJrbmQ==:bxQo+zXfjUgrD0jHuht0mQ=="
    }
}
```

```
{
    "class": "SensitiveUser\\User\\Domain\\Event\\UserRegistered",
    "payload": {
        "email": "#-#jTYqDtzJ8HHabEnJMMtuaiwiFcmCkZzel5985nSf\/
↪Ig=:iEMqT4YFE7OQzKdClNaDUg==",
        "id": "96607c7a-f4cd-4dd7-a406-9cde00913f79",
        "name": "Dario",
        "occurred_at": "2022-01-14T15:04:58.323+00:00",
        "surname": "#-#SXZXQsvLTCVX8Kel0yaoHg==:iEMqT4YFE7OQzKdClNaDUg=="
    }
}
```

```
{
    "class": "SensitiveUser\\User\\Domain\\Event\\UserRegistered",
    "payload": {
        "id": "c9298698-b30e-40c5-8d85-624fdf57f9df",
        "occurred_at": "2022-01-08T14:26:39.483+00:00",
        "name": "Matteo",
        "surname": "Galacci",
        "email": "#-
↪#aw+tw7shnEs2px030QS9WgRmGZckEGnIeR0a8ByMkPI=:Q0jkEOZtOs56tMkc8SjP5g=="
    }
}
```

1.3.4 Double key encryption

As mentioned above, when a new Aggregate is created, its key is also created and persisted in the appropriate table. Each aggregate has its own key so that it can invalidate individual Aggregates upon request. To improve security, the key of the aggregate, which we will call `AGGREGATE_KEY`, is in turn encrypted with what we will call `AGGREGATE_MASTER_KEY`.

AGGREGATE_KEY

- It persisted in the database and is in a 1:1 relationship with the aggregate.
- It is encrypted with the `AGGREGATE_MASTER_KEY`. This is to prevent events from being decrypted following a database violation.
- It can be deleted so as to make the Aggregate no longer decryptable

AGGREGATE_MASTER_KEY

- It is one for all `AGGREGATE_KEY`.
- It is not persisted in the database. It is set in an environment variable or otherwise on the server. More drivers will be available in the future to get the key.

1.4 Modules

The library consists of 2 Modules, `DataManager` and `Serializer`.

1.4.1 Architectural diagram

Fig. 3: Broadway sensitive serializer architecture

1.4.2 Data manager

`DataManager` module deals with data encryption and decryption, the creation of the `AGGREGATE_KEY` and the orchestration of the logics related to the sensitization and desensitization of events.

SensitiveDataManager

It is the interface for string encryption and decryption services. It asks for the implementation of the `doEncrypt` and `doDecrypt` methods, in which to implement your own concrete logic. The interface also provides the public constant SensitiveDataManager::IS_SENSITIZED_INDICATOR which is used as a prefix in encrypted strings in order to understand if a string is in the clear or not. This check can be done with the SensitiveTool::isSensitized(string $data): bool tool. Very convenient when it is necessary to carry out validations in the hydration phase, for example of a Value Object or an Aggregate.

The library provides an implementation of this interface that uses the AES256 algorithm: AES256SensitiveDataManager

KeyGenerator

It is the interface for the `AGGREGATE_KEY` creation services. Asks for the implementation of the `generate` method.

The library provides an implementation of this interface based on openssl: OpenSSLKeyGenerator

AggregateKeys

It is the interface to the repository that takes care of the persistence of the `AGGREGATE_KEY` through Model AggregateKey. It asks for the implementation of the `add`, `withAggregateId` and `update` methods.

A DBAL-based implementation is available by installing the Broadway Sensitive Serializer DBAL library.

1.4.3 Serializer

BroadwaySerializerDecorator

It is the abstract class that represents the original Broadway serializer decorator. It implements the Broadway's serializer interface and depends on an implementation of Broadway's serializer.

SensitiveSerializer

It is the concrete serializer implemented by the library. Extends BroadwaySerializerSerializer and depends on a BroadwaySerializerSerializer object (you can pass the standard Broadway serializer, SimpleInterfaceSerializer) and a SensitizerStrategy object.

1.4.4 Sensitization strategies

The library provides three different types of sensitization for the events payload, `Whole`, `Partial` and `Custom`.

Whole strategy

The Whole strategy aims to encrypt all the keys of the event payload with the exception of the aggregate id and the date of issue of the event.

```
{
    "class": "SensitiveUser\\User\\Domain\\Event\\UserRegistered",
    "payload": {
        "email": "#-
↪#0FLfN9XDKtWrmCmUb6mhY0Iz2V6wtam0pcqs6vDJFRU=:bxQo+zXfjUgrD0jHuht0mQ==",
        "id": "b0fce205-d816-46ac-886f-06de19236750",
        "name": "#-#EXWLg\/JANMK\/M+DmlpnOyQ==:bxQo+zXfjUgrD0jHuht0mQ==",
        "occurred_at": "2022-01-08T14:25:13.483+00:00",
        "surname": "#-#2Iuofg4NKKPLAG2kdJrbmQ==:bxQo+zXfjUgrD0jHuht0mQ=="
    }
}
```

The reference class for this strategy is WholePayloadSensitizer. While the client class of the strategy is WholeStrategy. This class depends on the `WholePayloadSensitizer` and the WholePayloadSensitizerRegistry registry which must be initialized with a `class-string[]` containing the list of FQCN (Full Qualified Class Name) of the events that you want to make subject to encryption. This therefore implies that not all events will be encrypted, but it can be selected selectively by populating the register.

Keys exclusion

The `id` key of the Aggregate can be configured during strategy creation via the WholePayloadSensitizer::$excludedIdKey attribute. In the same way it is possible to indicate a list of keys to be excluded from encryption using the WholePayloadSensitizer::$excludedKeys attribute.

Run whole strategy example example/WholeStrategy

```
make build-php ARG="--no-cache"
make upd
make composer ARG="install"
make enter
php example/WholeStrategy/example.php
```

Partial strategy

The partial strategy, probably the most convenient, involves the selective and parameterized encryption of a payload. It will be sufficient to pass to the PartialPayloadSensitizerRegistry register an array with the events to be encrypted and for each event, indicating the keys:

The client class of the strategy is PartialStrategy which is dependent on the `PartialPayloadSensitizerRegistry` and PartialPayloadSensitizer.

```
$events = [
   MyEvent::class => ['email', 'surname'],
   MySecondEvent::class => ['address'],
];

new PartialPayloadSensitizerRegistry($events);
```

Run partial strategy example example/PartialStrategy

```
make build-php ARG="--no-cache"
make upd
make composer ARG="install"
make enter
php example/PartialStrategy/example.php
```

Custom strategy

The Custom strategy involves the creation of specific `Sensitizers` in order to sensitize only a part of the payload according to the needs. These Sensitizers extend the abstract class PayloadSensitizer which involves the implementation of the PayloadSensitizer::generateSensitizedPayload(): array and PayloadSensitizer::generateDesensitizedPayload(): array methods.

Once defined, the Sensitizers must be used to initialize the specific CustomPayloadSensitizerRegistry registry of this strategy.

The client class of the strategy is CustomStrategy which is solely dependent on the `CustomPayloadSensitizerRegistry`. An example of implementation is present in the test.

Run custom strategy example example/CustomStrategy

```
make build-php ARG="--no-cache"
make upd
make composer ARG="install"
make enter
php example/CustomStrategy/example.php
```

Strategy summary

- With the Whole Strategy you can decide what not to encrypt if necessary, but not for a single event; you can exclude keys for all events subject to sensitization.
- With the Partial Strategy you define the events you want to encrypt, and for each event you define the list of keys to be excluded, using a simple array.
- With the Custom Strategy you have full control over how to intervene on the payload.

1.4.5 Value Serializer

PayloadSensitizer uses a value serializer respecting the ValueSerializer interface. This Serializer implements `strategy pattern` to be able to chose which type of serialization use. Broadway sensitive serializer provides a JsonValueSerializer implementation to serialize this types: `scalar`, `null`, `array`

1.4.6 AggregateKey model creation

The PayloadSensitizer::$automaticAggregateKeyCreation parameter determines if the AggregateKey model should be created automatically at serialization, or if you want to create it manually. The existence check of the model is not carried out in the PayloadSensitizer::desensitize(array $serializedObject): array method as it would be a contradiction; the process of saving events starts with the saving and relative serialization of a first event, so when calling the desensitize method it is assumed that the AggregateKey has already been created. Otherwise an exception will be throw.

Automatic creation

In this mode the `AggregateKey` model, if it does not exist, is created when calling method PayloadSensitizer::sensitize(array $serializedObject): array. The key is created if it does not exist, otherwise it uses the existing one:

```
$decryptedAggregateKey = $this->automaticAggregateKeyCreation ?
   $this->createAggregateKeyIfDoesNotExist($aggregateId) :
   $this->obtainDecryptedAggregateKeyOrError($aggregateId);
```

Manual creation

In this mode the `AggregateKey` model must exist, if it doesn't, an exception will be raised. This mode involves creating the model in advance. The most convenient time may be during the creation of the Aggregate.

```
$aggregateKeyManager->createAggregateKey($userId);

$user = User::create($userId, $name, $surname, $email, $registrationDate);

$users->add($user);
```

Generally speaking, the correct way to handle this in both ways would be to run the domain service atomically, within a transaction. The ddd-starter-pack library provide some convenient abstractions to handle this: TransactionalApplicationServiceTest

1.5 Examples

1.5.1 Library examples

In this repository you can find three example

- *Whole strategy*
- *Partial strategy*
- *Custom strategy*

Of course, you will also find many ideas in the tests.

Whole sensitization example

example/WholeStrategy

```
make build-php ARG="--no-cache"
make upd
make composer ARG="install"
make enter
php example/WholeStrategy/example.php | jq
```

Partial sensitization example

example/PartialStrategy

```
make build-php ARG="--no-cache"
make upd
make composer ARG="install"
make enter
php example/PartialStrategy/example.php | jq
```

Custom sensitization example

example/CustomStrategy

```
make build-php ARG="--no-cache"
make upd
make composer ARG="install"
make enter
php example/CustomStrategy/example.php | jq
```

1.5.2 Demo project

For a complete and working demo you can check out at this Symfony 6 project: broadway-sensitive-serializer-demo. It is divided into three branches:

- whole_strategy
- partial_strategy
- custom_strategy

1.6 Limitations

1.6.1 Existing CQRS + ES projects

If you have read the previous pages you will have understood that the idea behind this library is to create sensitized events from the beginning. Obviously I am not referring to all events, but to those containing sensitive data. If you have existing projects, with an Event Store that contains events where sensitive data is in the clear, as we said, creating compensation events will not serve you as the story remains clear. If you find yourself in this situation, the only way I can advise you for now, with all the relative contraindications, is to take all your events and migrate them to a new Event Store by execute a sensitization action in the middle, via the *Data manager* module. By doing this you will have a new Event Store, the same as the old one, but with encrypted sensitive data. From now on you will be able to use the library normally for the new events that will be generated.

Future idea

One idea for the future is to create a migration module in order to simplify the idea discussed above. With a simple configuration, you could automate the creation of a new Event Store.

```
$eventsToSensitise = [
    UserRegistered::class => [
        'email',
        'surname',
    ],
    PersonalDataAdded::class => [
        'religion',
    ]
];

$eventStoreMigrator->execute($eventsToSensitise);
```