Alaric

Skelmis

Dec 10, 2023

PRIMARY USAGE:

1 Main Usage **3**

2 Why Alaric **9**
- 2.1 TL;DR . . . 9
- 2.2 Basic Queries . . . 9
- 2.3 Automatic Class Conversion . . . 9
- 2.4 Advanced Querying . . . 11

3 Cursors **13**

4 Advanced Query Reference **17**

5 Comparators **19**
- 5.1 Equality . . . 19
- 5.2 Contains . . . 19
- 5.3 Existence . . . 20
- 5.4 Greater then . . . 20
- 5.5 Less then . . . 21

6 Logical Combinators **23**
- 6.1 Or . . . 23
- 6.2 And . . . 23
- 6.3 Not . . . 24

7 Meta Classes **25**
- 7.1 Negate . . . 25
- 7.2 All . . . 26

8 Projections **27**
- 8.1 Projection . . . 27
- 8.2 Show . . . 27
- 8.3 Hide . . . 28

9 Cached Documents **29**

10 Encryption at rest **31**
- 10.1 Database design impacts . . . 31
- 10.2 Class Reference . . . 32

11 Encryption Field Reference **39**

12 Protocols **41**

13 LogicalT Reference **43**

14 ComparisonT Reference **45**

15 Random class references **47**

16 Object Id **49**
16.1 ObjectId 49

17 Indices and tables **51**

Index **53**

Providing a beautiful way to interact with MongoDB asynchronously in Python.

```
pip install alaric
```

CHAPTER

ONE

MAIN USAGE

This is the most general use-case for Alaric, where-in you use the `Document` class.

class alaric.**Document**(*database: AsyncIOMotorDatabase*, *document_name: str*, *converter: Type[T] | None = None*)

__init__(*database: AsyncIOMotorDatabase*, *document_name: str*, *converter: Type[T] | None = None*)

Parameters

- **database** (*AsyncIOMotorDatabase*) – The database we are connected to
- **document_name** (*str*) – What this _document should be called
- **converter** (Optional[Type[*T*]]) – An optional class to try to convert all data-types which return either Dict or List into

```
from motor.motor_asyncio import AsyncIOMotorClient

client = AsyncIOMotorClient(connection_url)
database = client["my_database"]
config_document = Document(database, "config")
```

async bulk_insert(*data: List[Dict]*) → None

Given a List of Dictionaries, bulk insert all the given dictionaries in a single call.

Parameters

data (*List[Dict]*) – The data to bulk insert

```
# Insert 25 documents
await Document.bulk_insert(
    {"_id": i}
    for i in range(25)
)
```

async change_field_to(*filter_dict: Dict[str, Any] | Buildable | Filterable*, *field: str*, *new_value: Any*) → None

Modify a single field and change the value.

Parameters

- **filter_dict** (*Union[Dict[Any, Any], Buildable, Filterable]*) – A dictionary to use as a filter or *AQ* object.
- **field** (*str*) – The key for the field to increment
- **new_value** (*Any*) – What the field should get changed to

```python
# Assuming a data structure of
# {"_id": 1, "prefix": "!"}
await Document.change_field_to({"_id": 1}, "prefix", "?")

# This will now look like
# {"_id": 1, "prefix": "?"}
```

property collection_name: str

The connected collections name.

async count(*filter_dict: Dict[Any, Any]* | Buildable | Filterable) → int

Return a count of how many items match the filter.

Parameters
filter_dict (*Union[Dict[Any, Any], Buildable, Filterable]*) – The count filer.

Returns
How many items matched the filter.

Return type
int

```python
# How many items have the `enabled` field set to True
count: int = await Document.count({"enabled": True})
```

create_cursor() → *Cursor*

async delete(*filter_dict: Dict* | Buildable | Filterable) → DeleteResult | None

Delete an item from the Document if an item with the provided filter exists.

Parameters
filter_dict (*Union[Dict, Buildable, Filterable]*) – A dictionary to use as a filter or *AQ* object.

Returns
The result of deletion if it occurred.

Return type
Optional[DeleteResult]

```python
# Delete items with a `prefix` of `!`
await Document.delete({"prefix": "!"})
```

async delete_all() → None

Delete all data associated with this document.

Notes

This will attempt to complete the operation in a single call, however, if that fails it will fall back to manually deleting items one by one.

> **Warning:** There is no going back if you call this accidentally.
>
> This also currently doesn't appear to work.

property document_name: str

Same as *collection_name()*

async find(*filter_dict: Dict[str, Any] | Buildable | Filterable*, *projections: Dict[str, Any] | Projection | None = None*, ***, *try_convert: bool = True*) → Dict[str, Any] | Type[T] | None

Find and return one item.

Parameters

- **filter_dict** (*Union[Dict, Buildable, Filterable]*) – A dictionary to use as a filter or *AQ* object.
- **projections** (*Optional[Union[Dict[str, Any], Projection]]*) – Specify the data you want returned from matching queries.
- **try_convert** (*bool*) – Whether to attempt to run convertors on returned data.

  Defaults to True

Returns

The result of the query

Return type

Optional[Union[Dict[str, Any], Type[*T*]]]

```
# Find the document where the `_id` field is equal to `my_id`
data: dict = await Document.find({"_id": "my_id"})
```

async find_many(*filter_dict: Dict[str, Any] | Buildable | Filterable*, *projections: Dict[str, Any] | Projection | None = None*, ***, *try_convert: bool = True*) → List[Dict[str, Any] | Type[T]]

Find and return all items matching the given filter.

Parameters

- **filter_dict** (*Union[Dict[str, Any], Buildable, Filterable]*) – A dictionary to use as a filter or *AQ* object.
- **projections** (*Optional[Union[Dict[str, Any], Projection]]*) – Specify the data you want returned from matching queries.
- **try_convert** (*bool*) – Whether to attempt to run convertors on returned data.

  Defaults to True

Returns

The result of the query

Return type

List[Union[Dict[str, Any], Type[*T*]]]

Notes

This uses a cursor internally, consider using them for more complicated queries.

```
# Find all documents where the key `my_field` is `true`
data: list[dict] = await Document.find_many({"my_field": True})
```

async get_all(*filter_dict: Dict[str, Any] | Buildable | Filterable | None = None*, *projections: Dict[str, Any] | Projection | None = None*, **args: Any*, *try_convert: bool = True*, ***kwargs: Any*) → List[Dict[str, Any] | Type[T] | None]

Fetches and returns all items which match the given filter.

Parameters

- **filter_dict** (*Optional[Union[Dict[str, Any], Buildable, Filterable]]*) – A dictionary to use as a filter or *AQ* object.
- **projections** (*Optional[Union[Dict[str, Any], Projection]]*) – Specify the data you want returned from matching queries.
- **try_convert** (*bool*) – Whether to attempt to run convertors on returned data.

  Defaults to True

Returns
The items matching the filter

Return type
List[Optional[Union[Dict[str, Any], Type[*T*]]]]

```
data: list[dict] = await Document.get_all()
```

async increment(*filter_dict: Dict[str, Any] | Buildable | Filterable*, *field: str*, *amount: int | float*) → None

Increment the provided field.

Parameters

- **filter_dict** (*Union[Dict[str, Any], Buildable, Filterable]*) – The 'thing' we want to increment
- **field** (*str*) – The key for the field to increment
- **amount** (*Union[int, float]*) – How much to increment (or decrement) by

```
# Assuming a data structure of
# {"_id": 1, "counter": 4}
await Document.increment({"_id": 1}, "counter", 1)

# Now looks like
# {"_id": 1, "counter": 5}
```

Notes

You can also use negative numbers to decrease the count of a field.

async insert(*data: Dict[str, Any] | Saveable*) → None

Insert the provided data into the document.

Parameters
data (*Union[Dict[str, Any], Saveable]*) – The data to insert

```
# If you don't provide an _id,
# Mongo will generate one for you automatically
await Document.insert({"_id": 1, "data": "hello world"})
```

property raw_collection: AsyncIOMotorCollection

The connection collection instance.

property raw_database: AsyncIOMotorDatabase

Access to the database instance.

async unset(*filter_dict: Dict[str, Any]* | Buildable | Filterable, *field: Any*) → None

Delete a given field on a collection

Parameters

- **filter_dict** (*Union[Dict[str, Any], Buildable, Filterable]*) – The fields to match on (Think _id)
- **field** (*Any*) – The field to remove

```
# Assuming we have a document that looks like
# {"_id": 1, "field_one": True, "field_two": False}
await Document.unset({"_id": 1}, "field_two")

# This data will now look like the following
# {"_id": 1, "field_one": True}
```

async update(*filter_dict: Dict[str, Any]* | Buildable | Filterable, *update_data: Dict[str, Any]* | Saveable, *option: str = 'set'*, **args: Any*, ***kwargs: Any*) → None

Performs an UPDATE operation.

Parameters

- **filter_dict** (*Union[Dict[str, Any], Buildable, Filterable]*) – The data to filter on
- **update_data** (*Union[Dict[str, Any], Saveable]*) – The data to upsert
- **option** (*str*) – The optional option to pass to mongo, default is set

  https://www.mongodb.com/docs/manual/reference/operator/update/

```
# Update the document with an `_id` of 1
# So that it now equals the second argument
await Document.update({"_id": 1}, {"_id": 1, "data": "new data"})
```

async upsert(*filter_dict: Dict[str, Any]* | Buildable | Filterable, *update_data: Dict[str, Any]* | Saveable, *option: str = 'set'*, **args: Any*, ***kwargs: Any*) → None

Performs an UPSERT operation.

Parameters

- **filter_dict** (*Union[Dict[str, Any], Buildable, Filterable]*) – The data to filter on
- **update_data** (*Union[Dict[str, Any], Saveable]*) – The data to upsert
- **option** (*str*) – Update operator.

  https://www.mongodb.com/docs/manual/reference/operator/update/

```
# Update the document with an `_id` of `1`
# So that it now equals the second argument
# NOTE: If a document with an `_id` of `1`
# does not exist, then this method will
# insert the data instead.
await Document.upsert({"_id": 1}, {"_id": 1, "data": "new data"})
```

CHAPTER

TWO

WHY ALARIC

2.1 TL;DR

- Comes with typing and autocomplete support built in
- Support for class based structures, no more `data["_id"]` or `find_one({...})`
- Simplistic advanced querying support
- Convenience methods for everyday operations
- Built on motor, so if Alaric can't do it your not left in the lurch

---

For the sake of all examples, `document` will refer to an instance of *`Document`* while `collection` will refer to an instance of `AsyncIOMotorCollection`

2.2 Basic Queries

At a basic level, *`Document`* is used fairly similar to `AsyncIOMotorCollection` in that your basic queries cross over, for example:

```
# These are the same
r_1 = await collection.find_one({"_id": 1234})
r_2 = await document.find({"_id": 1234})
```

At its core, all methods accept the dictionaries you are used to. Where this is not true, your type checker will notify you.

2.3 Automatic Class Conversion

Lets imagine our data is structured like this:

```
{
    "_id": int,
    "prefix:" str,
    "activated_premium": bool,
}
```

With motor the following would be a fairly standard interaction.

```
data = await collection.find_one({"_id": 1234})
prefix = data["prefix"]
...
if data["activated_premium"]:
    ...
```

Raw dictionaries, no autocomplete, frankly yuck.

Now, by default Alaric also returns raw dictionaries, however, the following is also valid syntax and how I personally like to use the package.

```
class Guild:
    def __init__(self, _id, prefix, activated_premium):
        self._id: int = _id
        self.prefix: str = prefix
        self.activated_premium: bool = activated_premium

document = Document(..., converter=Guild)
guild: Guild = await document.find({"_id": 1234})
prefix = guild.prefix
...
if guild.activated_premium:
    ...
```

Note: Due to how Alaric is built, your `__init__` must accept all arguments from your returned data. If there is extra, say an unwanted `_id` I recommend just adding `**kwargs` and not using them in the method itself.

2.3.1 Fully utilizing class support

In the previous example you still have to convert it to a dictionary whenever you wanted to insert / update / filter. Lets change that.

Alaric exposes two protocol methods, which if implemented will be used.

These are:

- **`as_filter`**
  Treat the dictionary returned from this as a filter for a query.

  I.e. `as_filter` would return `{"_id": 1234}`

- **`as_dict`**
  Treat the dictionary returned from this as a full representation of the current object instance.

  I.e. `as_dict` would return `{"_id": 1234, "prefix": "!", "activated_premium": True}`

Lets see them in action.

```
from typing import Dict

class Guild:
    def __init__(self, _id, prefix, activated_premium):
        self._id: int = _id
        self.prefix: str = prefix
```

(continues on next page)

(continued from previous page)

```
        self.activated_premium: bool = activated_premium

    def as_filter(self) -> Dict:
        return {"_id": self._id}

    def as_dict(self) -> Dict:
        return {
            "_id": self._id,
            "prefix": self.prefix,
            "activated_premium": self.activated_premium,
        }

document = Document(..., converter=Guild)
guild: Guild = Guild(5678, "py.", False)
await document.insert(guild)

# Alternatively
guild: Guild = await document.find({"_id": 1234})
guild.prefix = "?"
await document.upsert(guild, guild)
```

Note: For the last example you should actually use *alaric.Document.change_field_to()*

2.3.2 Conditional Class Returns

In a situation where you don't want you returned data to be converted to your class?

Simply pass `try_convert=False` to the method.

2.4 Advanced Querying

This is a hidden gem, but MongoDB actually supports some extremely powerful queries. The issue however is the relevant dictionaries get big, quick.

Using our prior data structure, lets run a query to return all guilds that have the prefix `?`.

```
await document.find({"prefix": "?"})
```

Simple right?

How about all guilds where the prefix is either `!` or `??`

Now, the raw query for this would look something like this.

```
await document.find({'prefix': {'$in': ['!', '?']}})
```

But with Alaric you can make the same query like this.

```
from alaric import AQ
from alaric.comparison import IN
```

(continues on next page)

(continued from previous page)

```
await document.find(AQ(IN("prefix", ["!", "?"])))
```

I know what I'd prefer.

But lets make it even more complex!

Lets query for all the guilds that have activated premium, and have a prefix as either `!` or `?`.

Now, the raw query for this would look something like this.

```
await document.find(
    {
        "$and": [
            {"prefix": {"$in": ["!", "?"]}},
            {"activated_premium": {"$eq": True}},
        ]
    }
)
```

But with Alaric you can make the same query like this.

```
from alaric import AQ
from alaric.logical import AND
from alaric.comparison import EQ, IN

await document.find(AQ(AND(IN("prefix", ["!", "?"]), EQ("activated_premium", True))))
```

And this is only the tip of the iceberg, there are so many types of queries you can do.

CHAPTER

THREE

CURSORS

Cursor's allow for receiving more then 1 document at a time from your database.

class alaric.**Cursor**(*collection: AsyncIOMotorCollection*, ***, *converter: Type[C] | None = None*, *encryption_key: bytes | None = None*, *encrypted_fields:* EncryptedFields *| None = None*, *automatic_hashed_fields:* AutomaticHashedFields *| None = None*)

__aiter__()

This style of iteration is also supported.

```
cursor: Cursor = ...
async for entry in cursor:
    print(entry)
```

__init__(*collection: AsyncIOMotorCollection*, ***, *converter: Type[C] | None = None*, *encryption_key: bytes | None = None*, *encrypted_fields:* EncryptedFields *| None = None*, *automatic_hashed_fields:* AutomaticHashedFields *| None = None*)

Parameters

- **collection** (*AsyncIOMotorCollection*) – The motor collection
- **converter** (Optional[Type[C]]) – An optional class to try to convert all data-types which return either Dict or List into
- **encrypted_fields** (*Optional[EncryptedFields]*) – A list of fields to AES decrypt when encountered
- **encryption_key** (*Optional[bytes]*) – The key to use for AES decryption
- **automatic_hashed_fields** (*Optional[AutomaticHashedFields]*) – A list of fields to create an additional column in the db for with a hashed variant without exposing the hashed data to the end user.

Notes

This class is iterable using `async for`

copy() → *Cursor*

Returns a modifiable version of this cursor.

Returns
A new cursor instance

Return type
Cursor

async execute() → List[Dict[str, Any] | Type[C]]

Execute this cursor and return the result.

classmethod from_document(*document:* Document) → *Cursor*

Create a new *Cursor* from a *Document*

set_filter(*filter_data: Dict[str, Any]* | Buildable | Filterable = *ALL()*) → *Cursor*

Set the filter for the cursor query.

Parameters
filter_data (*Union[Dict[str, Any], Buildable, Filterable]*) – A dictionary to use as a filter or *AQ* object.

Returns
This cursor instance for method chaining.

Return type
Cursor

set_limit(*limit: int = 0*) → *Cursor*

Set a limit for how many documents should be returned in the query.

Use **0** to indicate no limit.

Parameters
limit (*int*) – How many documents should be returned.

Defaults to no limit.

Returns
This cursor instance for method chaining.

Return type
Cursor

Raises
ValueError – You specified a negative number.

set_projections(*projections: Dict[str, Literal[0, 1]]* | Projection | *None* = *None*) → *Cursor*

Define what data should be returned for each document in the result

Parameters
projections (*Optional[Union[Dict[str, Literal[0, 1]], Projection]]*) – Specify the data you want returned from matching queries.

Returns
This cursor instance for method chaining.

Return type
Cursor

set_sort(*order: List[Tuple[str, Any]]* | *Tuple[str, Any]* | *None* = *None*) → *Cursor*

Set the sort order for returned results

Parameters
order (*Optional[List[Tuple[str, Any]], Tuple[str, Any]]*) – The order to sort by

Returns
This cursor instance for method chaining

Return type
Cursor

Raises

- `ValueError` – The passed value was not a list or tuple
- `.. code-block:` – python: :linenos:

  import alaric

  # Lets sort by the count field Cursor.set_sort((“count”, alaric.Ascending))

  … # Lets sort first by the count field, # then also by the backup_count field Cursor.set_sort([(“count”, alaric.Ascending), (“backup_count”, alaric.Descending)])

CHAPTER

FOUR

ADVANCED QUERY REFERENCE

All requests using any query classes should be wrapped in an instance of this object. This tells the library that the query you are attempting to run requires conversion to something MongoDB will understand.

class alaric.**AQ**(*item:* ComparisonT | LogicalT | Buildable)

A container representing an advanced query.

Parameters

item (*Union[ComparisonT, LogicalT]*) – The parent item we wish to build upon.

```
# A query to fetch all items
# where the `id` field is equal to `1`
# AND the document contains a `prefix` field
from alaric import AQ
from alaric.logical import AND
from alaric.comparison import EQ, EXISTS

query = AQ(AND(EQ("id", 1), EXISTS("prefix")))
```

build() → Dict

Return this AQ as a usable Mongo filter.

CHAPTER

FIVE

COMPARATORS

This shows all the comparison objects and usages.

All of these classes are importable from `alaric.comparison`

5.1 Equality

class alaric.comparison.**EQ**(*field: str*, *value: int | str | float | bytes | dict | ObjectId*)

Asserts the provided field is equal to the provided value.

Parameters

- **field** (*str*) – The field to check in.
- **value** (*Union[int, str, float, bytes, dict, ObjectId]*) – The value the field should equal.

Notes

This also works on matching items in arrays in an OR based matching approach.

Lets match a document with an `_id` equal to 1

```
from alaric import AQ
from alaric.comparison import EQ

query = AQ(EQ("_id", 1))
```

build() → Dict[str, Dict[str, int | str | float | bytes | dict | *ObjectId*]]

Return this instance as a usable Mongo filter.

5.2 Contains

class alaric.comparison.**IN**(*field: str*, *value: list | tuple | set*)

Asserts the provided field is within the provided values.

This class can be used in conjunction with `alaric.meta.NEGATE`

Parameters

- **field** (*str*) – The field to check in.

- **value** (*Union[list, tuple, set]*) – A iterable of values that field should be in.

Lets match all documents where the field `counter` is equal to any number in this list.

```
from alaric import AQ
from alaric.comparison import IN

query = AQ(IN("counter", [1, 3, 5, 7, 9]))
```

build() → Dict[str, Dict[str, list | tuple | set]]

Return this instance as a usable Mongo filter.

5.3 Existence

Note: `EXISTS` is still exported for backwards compatibility reasons.

class alaric.comparison.**Exists**(*field: str*)

Returns all documents that contain this field.

This class can be used in conjunction with `NEGATE`

Parameters

field (*str*) – The field to check in.

Lets match all documents where the field `prefix` exists

```
from alaric import AQ
from alaric.comparison import EXISTS

query = AQ(EXISTS("prefix"))
```

build() → Dict[str, Dict[str, int | str | float | bytes]]

Return this instance as a usable Mongo filter.

5.4 Greater then

class alaric.comparison.**GT**(*field: str*, *value: int | str | float | bytes*)

Asserts the provided field is greater to the provided value.

This class can be used in conjunction with `NEGATE`

Parameters

- **field** (*str*) – The field to check in.
- **value** (*Union[int, str, float, bytes, dict]*) – The value the field should be greater than.

Lets match all documents where the field `counter` is greater then 5.

```
from alaric import AQ
from alaric.comparison import GT

query = AQ(GT("counter", 5))
```

build() → Dict[str, Dict[str, int | str | float | bytes]]

Return this instance as a usable Mongo filter.

5.5 Less then

class alaric.comparison.**LT**(*field: str*, *value: int | str | float | bytes*)

Asserts the provided field is less to the provided value.

This class can be used in conjunction with NEGATE

Parameters

- **field** (*str*) – The field to check in.
- **value** (*Union[int, str, float, bytes, dict]*) – The value the field should be less than.

Lets match all documents where the field counter is less then 5.

```
from alaric import AQ
from alaric.comparison import LT

query = AQ(LT("counter", 5))
```

build() → Dict[str, Dict[str, int | str | float | bytes]]

Return this instance as a usable Mongo filter.

CHAPTER

SIX

LOGICAL COMBINATORS

This shows all the logical objects and usages.

All of these classes are importable from `alaric.logical`

6.1 Or

class alaric.logical.**OR**(**comparisons:* ComparisonT | LogicalT | Buildable | *List[*ComparisonT | LogicalT | Buildable*]*)

Conduct a logical OR between all items passed to the constructor.

Lets build a check using a simple OR which will return all results the field `my_field` is either `True` OR a number in the list `[1, 2, 3, 7, 8, 9]`

```
from alaric import AQ
from alaric.logical import OR
from alaric.comparison import EQ, IN

query = AQ(OR(EQ("my_field", True), IN("my_field", [1, 2, 3, 7, 8, 9])))
```

build() → Dict[str, List[Dict]]

Return this instance as a usable Mongo filter.

6.2 And

class alaric.logical.**AND**(**comparisons:* ComparisonT | LogicalT | Buildable | *List[*ComparisonT | LogicalT | Buildable*]*)

Conduct a logical AND between all items passed to the constructor.

Lets build a query which returns all results where the field `discord` is equal to `Skelmis#9135` and `gamer_tag` is equal to `Skelmis`

```
from alaric import AQ
from alaric.logical import AND
from alaric.comparison import EQ

query = AQ(AND(EQ("discord", "Skelmis#9135"), EQ("gamer_tag", "Skelmis")))
```

build() → Dict[str, List[Dict]]

Return this instance as a usable Mongo filter.

6.3 Not

class alaric.logical.**NOT**(**comparisons:* ComparisonT | LogicalT | Buildable | *List[*ComparisonT | LogicalT | Buildable*]*)

Invert the effect of a query expression.

This back-links to here

Lets find all items where my `gamer_tag` does NOT match `Skelmis`

```
from alaric import AQ
from alaric.logical import NOT
from alaric.comparison import EQ

query = AQ(NOT(EQ("gamer_tag", "Skelmis")))
```

build() → Dict[str, List[Dict]]

Return this instance as a usable Mongo filter.

CHAPTER

SEVEN

META CLASSES

Extra classes which don't fit into a given category

All of these classes are importable from `alaric.meta`

7.1 Negate

Note: `NEGATE` is still exported for backwards compatibility reasons.

class alaric.meta.**Negate**(*comparison:* ComparisonT)

Negate a given option, I.e. Do the opposite.

Supported operands:

- *Exists*
- *IN*
- *GT*
- *LT*
- *EQ*

Lets get all documents *without* a field called `prefix`

```
from alaric.comparison import Exists
from alaric.meta import Negate
from alaric import AQ

query = AQ(Negate(Exists("prefix")))
```

build() → Dict

Returns a mongo usable filter for the negated option.

7.2 All

class alaric.meta.**All**

Return all documents in the collection.

```
from alaric.meta import All

query = AQ(All())
```

build() → Dict

CHAPTER

EIGHT

PROJECTIONS

Note: `PROJECTION`, `SHOW` and `HIDE` are still exported for backwards compatibility reasons.

All of these classes are importable from `alaric.projections`

8.1 Projection

class alaric.projections.**Projection**(**fields:* Show | Hide)

Specify that only the given fields should be returned from the query.

```
# Assuming the data structure
# {"_id": 1234, "prefix": "!", "has_premium": False}
data: dict = await Document.find(
    {"_id": "my_id"}, projections=Projection(Show("prefix"))
)
print(data)
# Will print {"prefix": "!"}
```

build() → Dict

8.2 Show

class alaric.projections.**Show**(**fields: Any*)

Show this field in the returned data.

build() → Dict

8.3 Hide

class alaric.projections.**Hide**(**fields: Any*)

Hide this field in the returned data.

build() → Dict

CHAPTER

NINE

CACHED DOCUMENTS

For read heavy workloads were you may wish to reduce database load, Alaric offers *alaric.CachedDocument* which implements a Redis based cache in front of your database automatically.

class alaric.**CachedDocument**(*, *document:* Document, *redis_client: Redis*, *extra_lookups: List[List[str]] = None*, *cache_ttl: timedelta = datetime.timedelta(seconds=3600)*)

This document implements a cache in front of MongoDB for read heavy work flows.

Read process:

```
result = attempt to hit redis
if result is None:
    result = fetch from database
    populate redis cache
```

Write process:

```
Push changes back to database
update redis cache
```

Notes

This document works off the assumption that a documents `_id` remains consistent through the lifetime of the entry.

This document will also leave hanging redis entries when lookups outside the _id are modified during the lifetime of the object. This is mitigated by enforcing a TTL of all redis entries.

__init__(*, *document:* Document, *redis_client: Redis*, *extra_lookups: List[List[str]] = None*, *cache_ttl: timedelta = datetime.timedelta(seconds=3600)*)

Parameters

- **document** (*alaric.Document*) – The underlying DB document
- **redis_client** (*redis.asyncio.client.Redis*) – The Redis instance to use
- **extra_lookups** (*List[List[str]]*) – Extra lookups to build. For example:

```
class Test:
    def __init__(self, _id, data):
        self.data = data
        self._id = _id
```

(continues on next page)

(continued from previous page)

```
...

extra_lookups = [["data"]]
```

Would let you use `get` with either the `_id` or `data` parameter of `Test`

- **cache_ttl** (*timedelta*) – How long keys should exist in Redis.

  This is a requirement as this class will leave hanging keys in Redis when certain values change.

async get(*filter_dict: Dict[str, Any] | Buildable | Filterable*, ***, *try_convert: bool = True*) → Dict[str, Any] | C | None

Fetch a document.

Attempts to fetch from Redis and then falls back to DB

Parameters

- **filter_dict** (*Union[Dict[str, Any], Buildable, Filterable]*) –
- **try_convert** (*bool*) – See *alaric.Document*

Returns
The data fetched from either Redis or your DB

Return type
Optional[Union[Dict[str, Any], C]]

Notes

Due to how items are stored internally, filter_dict must eval to a literal str: str pairing otherwise this will fall back to the DB

async set(*filter_dict: Dict[str, Any] | Buildable | Filterable*, *update_data: Dict[str, Any] | Saveable*) → None

Write to Redis and the DB.

Notes

This performs an UPSERT operation on the database.

This also requires `_id` to be set on update_data in order to cache this to Redis

Parameters

- **filter_dict** (*Union[Dict[str, Any], Buildable, Filterable]*) –
- **update_data** (*Union[Dict[str, Any], Saveable]*) –

CHAPTER

TEN

ENCRYPTION AT REST

Data security is a big thing these days and its especially prevalent in my life as I work in Cyber Security, however, I am also not a massive business who can afford to pay for Mongo enterprise to support encryption at rest so this is my next best thing.

The new document subclass supports both encrypting fields with AES and hashing fields with SHA512.

This does not replace password hashing. Do not use this for storing passwords. Use an algorithm such as Bcrypt or Argon2id

10.1 Database design impacts

Q: I want to encrypt my fields but need to be able to do query filtering on them?

A: I suggest mirroring the fields, one hashed and one encrypted.

You can run filters against the hashed field as hashes don't change, and when you need to gain access to that data you can fetch it via the encrypted field.

Note that if your database is leaked this does mean the barrier to retrieving your data is only that the hash is cracked. The hashing used in Alaric should not be considered secure against brute forcing methods and thus you accept that data may be reversed from hashed fields.

Q: I want to hash my `XXX` field, but I also need to know the value sometimes?

A: See above.

Q: How do I deal with encrypted items in cursor's?

A: Cursor's have first class support for encrypted fields.

Q: Why are only the field values encrypted?

A: Because this is driver support, not built in.

If you want to hide keys, consider nesting your data behind a single key which is encrypted. It's not a great idea, but it'd work.

Q: I want my *as_filter* method on my converter to be a hashed filter.

A: You can use *alaric.util.hash_field* to hash your returned dictionary values in the same way that Alaric will hash your DB. This means you can filter based on hashed fields easily.

For example:

```
from alaric import util

class Test:
    def __init__(self, data, id, _id=None):
        self.data = data
        self.id = id
        self._id = _id

    def as_dict(self):
        return {"data": self.data, "id": self.id}

    def as_filter(self):
        return {"id_hashed": util.hash_field("id", self.id)}
```

Q: I encrypted my data using a generated key and lost it. Help!

A: Your data is gone if you lose your key.

The whole point of encrypting fields is so people without the key are unable to decrypt the data. When you lose the key, you also fall into this group of people.

Q: How do I query a hashed field if I don't know the hash?

A: Created your query as per usual, just wrap your comparison object in `HQF(...)` and Alaric will handle it for you.

```
from alaric import AQ
from alaric.comparison import EQ
from alaric.encryption import HQF

query = AQ(HQF(EQ("_id", 1)))
```

10.2 Class Reference

class alaric.**EncryptedDocument**(*database: AsyncIOMotorDatabase*, *document_name: str*, ***, *encryption_key: bytes*, *hashed_fields:* HashedFields | *None* = *None*, *automatic_hashed_fields:* AutomaticHashedFields | *None* = *None*, *encrypted_fields:* EncryptedFields | *None* = *None*, *converter: Type[T]* | *None* = *None*, *encrypt_all_fields: bool* = *False*)

__init__(*database: AsyncIOMotorDatabase*, *document_name: str*, ***, *encryption_key: bytes*, *hashed_fields:* HashedFields | *None* = *None*, *automatic_hashed_fields:* AutomaticHashedFields | *None* = *None*, *encrypted_fields:* EncryptedFields | *None* = *None*, *converter: Type[T]* | *None* = *None*, *encrypt_all_fields: bool* = *False*)

Parameters

- **database** (*AsyncIOMotorDatabase*) – The database we are connected to
- **document_name** (*str*) – What this collection should be called
- **encryption_key** (*bytes*) – The key to use for AES encryption
- **hashed_fields** (*Optional[HashedFields]*) – A list of fields to SHA512 hash when encountered

- **automatic_hashed_fields** (*Optional[AutomaticHashedFields]*) – A list of fields to create an additional column in the db for with a hashed variant without exposing the hashed data to the end user.
- **encrypted_fields** (*Optional[EncryptedFields]*) – A list of fields to AES encrypt when encountered
- **converter** (Optional[Type[*T*]]) – An optional class to try to convert all data-types which return either Dict or List into
- **encrypt_all_fields** (*bool*) – If set to True, encrypt all fields regardless of *hashed_fields* and *encrypted_fields* options.

  This option respects ignored fields.

```
from motor.motor_asyncio import AsyncIOMotorClient

client = AsyncIOMotorClient(connection_url)
database = client["my_database"]
config_document = Document(database, "config")
```

async bulk_insert(*data: List[Dict]*, *ignore_fields: IgnoreFields | None = None*) → None

Given a List of Dictionaries, bulk insert all the given dictionaries in a single call.

Notes

Supports encrypted and hashed fields.

Parameters

- **data** (*List[Dict]*) – The data to bulk insert
- **ignore_fields** (*Optional[IgnoreFields]*) – Any fields to ignore during the hashing / encryption step.

  Useful if your passing this method an already hashed value and you don't want to hash the hash.

```
# Insert 25 documents
await Document.bulk_insert(
    {"_id": i}
    for i in range(25)
)
```

async change_field_to(*filter_dict: Dict[str, Any] | Buildable | Filterable*, *field: str*, *new_value: Any*) → None

Modify a single field and change the value.

Parameters

- **filter_dict** (*Union[Dict[Any, Any], Buildable, Filterable]*) – A dictionary to use as a filter or *AQ* object.
- **field** (*str*) – The key for the field to increment
- **new_value** (*Any*) – What the field should get changed to

```python
# Assuming a data structure of
# {"_id": 1, "prefix": "!"}
await Document.change_field_to({"_id": 1}, "prefix", "?")

# This will now look like
# {"_id": 1, "prefix": "?"}
```

property collection_name: str

The connected collections name.

async count(*filter_dict: Dict[Any, Any]* | Buildable | Filterable) → int

Return a count of how many items match the filter.

Parameters
filter_dict (*Union[Dict[Any, Any], Buildable, Filterable]*) – The count filer.

Returns
How many items matched the filter.

Return type
int

```python
# How many items have the `enabled` field set to True
count: int = await Document.count({"enabled": True})
```

create_cursor() → *Cursor*

async delete(*filter_dict: Dict* | Buildable | Filterable) → DeleteResult | None

Delete an item from the Document if an item with the provided filter exists.

Parameters
filter_dict (*Union[Dict, Buildable, Filterable]*) – A dictionary to use as a filter or *AQ* object.

Returns
The result of deletion if it occurred.

Return type
Optional[DeleteResult]

```python
# Delete items with a `prefix` of `!`
await Document.delete({"prefix": "!"})
```

async delete_all() → None

Delete all data associated with this document.

Notes

This will attempt to complete the operation in a single call, however, if that fails it will fall back to manually deleting items one by one.

> **Warning:** There is no going back if you call this accidentally.
>
> This also currently doesn't appear to work.

property document_name: *str*

Same as *collection_name()*

async find(*filter_dict: Dict[str, Any] | Buildable | Filterable*, *projections: Dict[str, Any] | Projection | None = None*, ***, *try_convert: bool = True*) → Dict[str, Any] | Type[T] | None

Find and return one item.

Parameters

- **filter_dict** (*Union[Dict, Buildable, Filterable]*) – A dictionary to use as a filter or *AQ* object.
- **projections** (*Optional[Union[Dict[str, Any], Projection]]*) – Specify the data you want returned from matching queries.
- **try_convert** (*bool*) – Whether to attempt to run convertors on returned data.

  Defaults to True

Returns

The result of the query

Return type

Optional[Union[Dict[str, Any], Type[*T*]]]

```
# Find the document where the `_id` field is equal to `my_id`
data: dict = await Document.find({"_id": "my_id"})
```

async find_many(*filter_dict: Dict[str, Any] | Buildable | Filterable*, *projections: Dict[str, Any] | Projection | None = None*, ***, *try_convert: bool = True*) → List[Dict[str, Any] | Type[T]]

Find and return all items matching the given filter.

Parameters

- **filter_dict** (*Union[Dict[str, Any], Buildable, Filterable]*) – A dictionary to use as a filter or *AQ* object.
- **projections** (*Optional[Union[Dict[str, Any], Projection]]*) – Specify the data you want returned from matching queries.
- **try_convert** (*bool*) – Whether to attempt to run convertors on returned data.

  Defaults to True

Returns

The result of the query

Return type

List[Union[Dict[str, Any], Type[*T*]]]

Notes

This uses a cursor internally, consider using them for more complicated queries.

```
# Find all documents where the key `my_field` is `true`
data: list[dict] = await Document.find_many({"my_field": True})
```

classmethod generate_aes_key() → bytes

Generate a valid AES key for usage with this class.

The output should be stored in an environment variable for future usage as otherwise you will lose your data.

For storage purposes, you may find the following methods useful:

- bytes.hex()
- bytes.fromhex()

Returns
A valid AES key

Return type
bytes

async get_all(*filter_dict: Dict[str, Any] | Buildable | Filterable | None = None*, *projections: Dict[str, Any] | Projection | None = None*, **args: Any*, *try_convert: bool = True*, ***kwargs: Any*) → List[Dict[str, Any] | Type[T] | None]

Fetches and returns all items which match the given filter.

Parameters

- **filter_dict** (*Optional[Union[Dict[str, Any], Buildable, Filterable]]*) – A dictionary to use as a filter or *AQ* object.
- **projections** (*Optional[Union[Dict[str, Any], Projection]]*) – Specify the data you want returned from matching queries.
- **try_convert** (*bool*) – Whether to attempt to run convertors on returned data.

  Defaults to True

Returns
The items matching the filter

Return type
List[Optional[Union[Dict[str, Any], Type[*T*]]]]

```
data: list[dict] = await Document.get_all()
```

async increment(*filter_dict: Dict[str, Any] | Buildable | Filterable*, *field: str*, *amount: int | float*) → None

Increment the provided field.

Parameters

- **filter_dict** (*Union[Dict[str, Any], Buildable, Filterable]*) – The 'thing' we want to increment
- **field** (*str*) – The key for the field to increment
- **amount** (*Union[int, float]*) – How much to increment (or decrement) by

Notes

This seamlessly handles incrementing encrypted fields.

```
# Assuming a data structure of
# {"_id": 1, "counter": 4}
await Document.increment({"_id": 1}, "counter", 1)

# Now looks like
# {"_id": 1, "counter": 5}
```

Raises

- **ValueError** – Nested field updates on encrypted fields is not supported.
- **ValueError** – Item to increment didn't exist with this filter.

Notes

You can also use negative numbers to decrease the count of a field.

async insert(*data: Dict[str, Any]* | Saveable, *, *ignore_fields: IgnoreFields* | *None* = *None*) → None

Insert the provided data into the document.

Parameters

- **data** (*Union[Dict[str, Any], Saveable]*) – The data to insert
- **ignore_fields** (*Optional[IgnoreFields]*) – Any fields to ignore during the hashing / encryption step.

  Useful if your passing this method an already hashed value and you don't want to hash the hash.
- **code-block:** (..) – python: :linenos:

  # If you don't provide an _id, # Mongo will generate one for you automatically await Document.insert({"_id": 1, "data": "hello world"})

property raw_collection: AsyncIOMotorCollection

The connection collection instance.

property raw_database: AsyncIOMotorDatabase

Access to the database instance.

async unset(*filter_dict: Dict[str, Any]* | Buildable | Filterable, *field: Any*) → None

Delete a given field on a collection

Parameters

- **filter_dict** (*Union[Dict[str, Any], Buildable, Filterable]*) – The fields to match on (Think _id)
- **field** (*Any*) – The field to remove

```
# Assuming we have a document that looks like
# {"_id": 1, "field_one": True, "field_two": False}
await Document.unset({"_id": 1}, "field_two")

```

(continues on next page)

(continued from previous page)

```
# This data will now look like the following
# {"_id": 1, "field_one": True}
```

async update(*filter_dict: Dict[str, Any] | Buildable | Filterable*, *update_data: Dict[str, Any] | Saveable*, *option: str = 'set'*, **args: Any*, *ignore_fields: IgnoreFields | None = None*, ***kwargs: Any*) → None

Performs an UPDATE operation.

Parameters

- **filter_dict** (*Union[Dict[str, Any], Buildable, Filterable]*) – The data to filter on
- **update_data** (*Union[Dict[str, Any], Saveable]*) – The data to upsert
- **option** (*str*) – The optional option to pass to mongo, default is set

  https://www.mongodb.com/docs/manual/reference/operator/update/
- **ignore_fields** (*Optional[IgnoreFields]*) – Any fields to ignore during the hashing / encryption step.

  Useful if your passing this method an already hashed value and you don't want to hash the hash.

```
# Update the document with an `_id` of 1
# So that it now equals the second argument
await Document.upsert({"_id": 1}, {"_id": 1, "data": "new data"})
```

async upsert(*filter_dict: Dict[str, Any] | Buildable | Filterable*, *update_data: Dict[str, Any] | Saveable*, *option: str = 'set'*, **args: Any*, *ignore_fields: IgnoreFields | None = None*, ***kwargs: Any*) → None

Performs an UPSERT operation.

Parameters

- **filter_dict** (*Union[Dict[str, Any], Buildable, Filterable]*) – The data to filter on
- **update_data** (*Union[Dict[str, Any], Saveable]*) – The data to upsert
- **option** (*str*) – Update operator.

  https://www.mongodb.com/docs/manual/reference/operator/update/
- **ignore_fields** (*Optional[IgnoreFields]*) – Any fields to ignore during the hashing / encryption step.

  Useful if your passing this method an already hashed value and you don't want to hash the hash.

```
# Update the document with an `_id` of `1`
# So that it now equals the second argument
# NOTE: If a document with an `_id` of `1`
# does not exist, then this method will
# insert the data instead.
await Document.update({"_id": 1}, {"_id": 1, "data": "new data"})
```

CHAPTER

ELEVEN

ENCRYPTION FIELD REFERENCE

Two classes used for denoting which fields to hash / encrypt

class alaric.encryption.**HashedFields**(**fields: str*)

A list of fields which should be hashed when encountered.

Due to the nature of hashing, this is a one way operation and is only done when sending items to Mongo.

Note: Take care not to pass the hashed output back to a method which sends data to Mongo as it will result in it being re-hashed.

If this is the case, tell the document method to ignore processing the field.

```
from alaric.encryption import HashedFields

HashedFields("guild_id", "test")
```

class alaric.encryption.**AutomaticHashedFields**(**fields: str*)

A list of fields which should have a hashed field created automatically in the background but without access to the user.

I.e. Alaric when told to automatically hash the field `guild_id` will look at incoming data, create an extra field called `guild_id_hashed` and add it to the database. Whenever you fetch data from the database, Alaric will remove this field so you never see it.

If you want to use this field in your code, use *alaric.encryption.HashedFields* instead.

```
from alaric.encryption import AutomaticHashedFields

AutomaticHashedFields("guild_id", "test")
```

Note: You can use this in conjunction with an encrypted field to have a hash representing the unencrypted data.

---

class alaric.encryption.**EncryptedFields**(**fields: str*)

A list of fields which should be encrypted when encountered.

```
from alaric.encryption import EncryptedFields

EncryptedFields("guild_id", "test")
```

class alaric.encryption.**HashedQueryField**(*entry:* ComparisonT)

Use this to query against hashed fields.

This class exposes an alias *HQF* for shorter usage.

```
from alaric import AQ
from alaric.comparison import EQ
from alaric.encryption import HQF

query = AQ(HQF(EQ("_id", 1)))
```

build() → Dict[str, Dict[str, int | str | float | bytes | dict | *ObjectId*]]

Return this instance as a usable Mongo filter.

alaric.util.**hash_field**(*field*, *value*)

Hash a field with SHA512

Parameters

- **field** (*str*) – The name of the field your hashing
- **value** – The value to hash

Raises

ValueError – Unsupported type to hash

CHAPTER

TWELVE

PROTOCOLS

class alaric.abc.**Buildable**(**args*, ***kwargs*)

Protocol for class based Queries.

build() → Dict

Return this instance as a usable Mongo filter.

class alaric.abc.**Filterable**(**args*, ***kwargs*)

Protocol for class based Queries.

as_filter() → Dict

Returns a dictionary that represents a filter required to return this object.

class alaric.abc.**Saveable**(**args*, ***kwargs*)

Protocol for class based Queries.

as_dict() → Dict

Returns this class represented as a dictionary.

class alaric.abc.**ComparisonT**(**args*, ***kwargs*)

A protocol for all Comparison classes to follow.

build() → Dict

Return this instance as a usable Mongo filter.

class alaric.abc.**LogicalT**(**args*, ***kwargs*)

A protocol for all Logical classes to follow.

build() → Dict[str, List[Dict]]

Return this instance as a usable Mongo filter.

CHAPTER

THIRTEEN

LOGICALT REFERENCE

class alaric.abc.**LogicalT**(**args*, ***kwargs*)

A protocol for all Logical classes to follow.

build() → Dict[str, List[Dict]]

Return this instance as a usable Mongo filter.

CHAPTER

FOURTEEN

COMPARISONT REFERENCE

class alaric.abc.**ComparisonT**(**args*, ***kwargs*)

A protocol for all Comparison classes to follow.

build() → Dict

Return this instance as a usable Mongo filter.

CHAPTER

FIFTEEN

RANDOM CLASS REFERENCES

alaric.document.**T**

Invariant TypeVar.

A typevar representing the type of a given converter class

alaric.cursor.**C**

Invariant TypeVar.

A typevar representing the type of a given converter class

CHAPTER

SIXTEEN

OBJECT ID

This class wraps the `bson.ObjectId` to provide a simplistic way to use native object id's within Alaric.

16.1 ObjectId

class alaric.types.**ObjectId**(*object_id: str* | ObjectId | *bytes*)

A thin wrapper over bson.ObjectId for usage within Alaric

property object_id: `str` | *`ObjectId`* | `bytes`

CHAPTER

SEVENTEEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

__aiter__() (*alaric.Cursor method*), 13
__init__() (*alaric.CachedDocument method*), 29
__init__() (*alaric.Cursor method*), 13
__init__() (*alaric.Document method*), 3
__init__() (*alaric.EncryptedDocument method*), 32

A

All (*class in alaric.meta*), 26
AND (*class in alaric.logical*), 23
AQ (*class in alaric*), 17
as_dict() (*alaric.abc.Saveable method*), 41
as_filter() (*alaric.abc.Filterable method*), 41
AutomaticHashedFields (*class in alaric.encryption*), 39

B

build() (*alaric.abc.Buildable method*), 41
build() (*alaric.abc.ComparisonT method*), 45
build() (*alaric.abc.LogicalT method*), 43
build() (*alaric.AQ method*), 17
build() (*alaric.comparison.EQ method*), 19
build() (*alaric.comparison.Exists method*), 20
build() (*alaric.comparison.GT method*), 21
build() (*alaric.comparison.IN method*), 20
build() (*alaric.comparison.LT method*), 21
build() (*alaric.encryption.HashedQueryField method*), 40
build() (*alaric.logical.AND method*), 23
build() (*alaric.logical.NOT method*), 24
build() (*alaric.logical.OR method*), 23
build() (*alaric.meta.All method*), 26
build() (*alaric.meta.Negate method*), 25
build() (*alaric.projections.Hide method*), 28
build() (*alaric.projections.Projection method*), 27
build() (*alaric.projections.Show method*), 27
Buildable (*class in alaric.abc*), 41
bulk_insert() (*alaric.Document method*), 3
bulk_insert() (*alaric.EncryptedDocument method*), 33

C

C (*in module alaric.cursor*), 47
CachedDocument (*class in alaric*), 29
change_field_to() (*alaric.Document method*), 3
change_field_to() (*alaric.EncryptedDocument method*), 33
collection_name (*alaric.Document property*), 4
collection_name (*alaric.EncryptedDocument property*), 34
ComparisonT (*class in alaric.abc*), 45
copy() (*alaric.Cursor method*), 13
count() (*alaric.Document method*), 4
count() (*alaric.EncryptedDocument method*), 34
create_cursor() (*alaric.Document method*), 4
create_cursor() (*alaric.EncryptedDocument method*), 34
Cursor (*class in alaric*), 13

D

delete() (*alaric.Document method*), 4
delete() (*alaric.EncryptedDocument method*), 34
delete_all() (*alaric.Document method*), 4
delete_all() (*alaric.EncryptedDocument method*), 34
Document (*class in alaric*), 3
document_name (*alaric.Document property*), 4
document_name (*alaric.EncryptedDocument property*), 34

E

EncryptedDocument (*class in alaric*), 32
EncryptedFields (*class in alaric.encryption*), 39
EQ (*class in alaric.comparison*), 19
execute() (*alaric.Cursor method*), 13
Exists (*class in alaric.comparison*), 20

F

Filterable (*class in alaric.abc*), 41
find() (*alaric.Document method*), 5
find() (*alaric.EncryptedDocument method*), 35
find_many() (*alaric.Document method*), 5
find_many() (*alaric.EncryptedDocument method*), 35
from_document() (*alaric.Cursor class method*), 14

G

generate_aes_key() (*alaric.EncryptedDocument class method*), 35
get() (*alaric.CachedDocument method*), 30
get_all() (*alaric.Document method*), 5
get_all() (*alaric.EncryptedDocument method*), 36
GT (*class in alaric.comparison*), 20

H

hash_field() (*in module alaric.util*), 40
HashedFields (*class in alaric.encryption*), 39
HashedQueryField (*class in alaric.encryption*), 39
Hide (*class in alaric.projections*), 28

I

IN (*class in alaric.comparison*), 19
increment() (*alaric.Document method*), 6
increment() (*alaric.EncryptedDocument method*), 36
insert() (*alaric.Document method*), 6
insert() (*alaric.EncryptedDocument method*), 37

L

LogicalT (*class in alaric.abc*), 43
LT (*class in alaric.comparison*), 21

N

Negate (*class in alaric.meta*), 25
NOT (*class in alaric.logical*), 24

O

object_id (*alaric.types.ObjectId property*), 49
ObjectId (*class in alaric.types*), 49
OR (*class in alaric.logical*), 23

P

Projection (*class in alaric.projections*), 27

R

raw_collection (*alaric.Document property*), 6
raw_collection (*alaric.EncryptedDocument property*), 37
raw_database (*alaric.Document property*), 6
raw_database (*alaric.EncryptedDocument property*), 37

S

Saveable (*class in alaric.abc*), 41
set() (*alaric.CachedDocument method*), 30
set_filter() (*alaric.Cursor method*), 14
set_limit() (*alaric.Cursor method*), 14
set_projections() (*alaric.Cursor method*), 14
set_sort() (*alaric.Cursor method*), 14
Show (*class in alaric.projections*), 27

T

T (*in module alaric.document*), 47

U

unset() (*alaric.Document method*), 7
unset() (*alaric.EncryptedDocument method*), 37
update() (*alaric.Document method*), 7
update() (*alaric.EncryptedDocument method*), 38
upsert() (*alaric.Document method*), 7
upsert() (*alaric.EncryptedDocument method*), 38