2.3. Defining documents

In MongoDB, a **document** is roughly equivalent to a **row** in an RDBMS. When working with relational databases, rows are stored in **tables**, which have a strict **schema** that the rows follow. MongoDB stores documents in **collections** rather than tables — the principal difference is that no schema is enforced at a database level.

2.3.1. Defining a document's schema

MongoEngine allows you to define schemata for documents as this helps to reduce coding errors, and allows for utility methods to be defined on fields which may be present.

To define a schema for a document, create a class that inherits from `Document`. Fields are specified by adding **field objects** as class attributes to the document class:

```
from mongoengine import *
import datetime

class Page(Document):
    title = StringField(max_length=200, required=True)
    date_modified = DateTimeField(default=datetime.datetime.utcnow)
```

As BSON (the binary format for storing data in mongodb) is order dependent, documents are serialized based on their field order.

2.3.2. Dynamic document schemas

One of the benefits of MongoDB is dynamic schemas for a collection, whilst data should be planned and organised (after all explicit is better than implicit!) there are scenarios where having dynamic / expando style documents is desirable.

`DynamicDocument` documents work in the same way as `Document` but any data / attributes set to them will also be saved

```
from mongoengine import *

class Page(DynamicDocument):
    title = StringField(max_length=200, required=True)

# Create a new page and add tags
>>> page = Page(title='Using MongoEngine')
>>> page.tags = ['mongodb', 'mongoengine']
>>> page.save()

>>> Page.objects(tags='mongoengine').count()
>>> 1
```

Note

There is one caveat on Dynamic Documents: fields cannot start with _

Dynamic fields are stored in creation order *after* any declared fields.

2.3.3. Fields

By default, fields are not required. To make a field mandatory, set the `required` keyword argument of a field to `True`. Fields also may have validation constraints available (such as `max_length` in the example above). Fields may also take default values, which will be used if a value is not provided. Default values may optionally be a callable, which will be called to retrieve the value (such as in the above example). The field types available are as follows:

`BinaryField`

`BooleanField`

`ComplexDateTimeField`

`DateTimeField`

`DecimalField`

`DictField`

`DynamicField`

`EmailField`

`EmbeddedDocumentField`

`EmbeddedDocumentListField`

`EnumField`

`FileField`

`FloatField`

`GenericEmbeddedDocumentField`

`GenericReferenceField`

`GenericLazyReferenceField`

`GeoPointField`

`ImageField`

`IntField`

`ListField`

`LongField`

`MapField`

`ObjectIdField`

`ReferenceField`

`LazyReferenceField`

`SequenceField`

`SortedListField`

`StringField`

`URLField`

`UUIDField`

`PointField`

`LineStringField`

`PolygonField`

`MultiPointField`

`MultiLineStringField`

`MultiPolygonField`

2.3.3.1. Field arguments

Each field type can be customized by keyword arguments. The following keyword arguments can be set on all fields:

`db_field` (Default: None)

The MongoDB field name.

If set, operations in MongoDB will be performed with this value instead of the class attribute.

This allows you to use a different attribute than the name of the field used in MongoDB.

```
from mongoengine import *

class Page(Document):
    page_number = IntField(db_field="pageNumber")

# Create a Page and save it
Page(page_number=1).save()

# How 'pageNumber' is stored in MongoDB
Page.objects.as_pymongo() # [{'_id': ObjectId('629dfc45ee4cc407b1586b1f'),
'pageNumber': 1}]

# Retrieve the object
page: Page = Page.objects.first()

print(page.page_number)  # prints 1

print(page.pageNumber) # raises AttributeError
```

Note

If set, use the name of the attribute when defining indexes in the `meta` dictionary rather than the `db_field` otherwise, `LookUpError` will be raised.

`required` (Default: False)

If set to True and the field is not set on the document instance, a `ValidationError` will be raised when the document is validated.

`default` (Default: None)

A value to use when no value is set for this field.

The definition of default parameters follow the general rules on Python, which means that some care should be taken when dealing with default mutable objects (like in `ListField` or `DictField`):

```
class ExampleFirst(Document):
    # Default an empty list
    values = ListField(IntField(), default=list)

class ExampleSecond(Document):
    # Default a set of values
    values = ListField(IntField(), default=lambda: [1,2,3])

class ExampleDangerous(Document):
    # This can make an .append call to  add values to the default (and all the
following objects),
    # instead to just an object
    values = ListField(IntField(), default=[1,2,3])
```

Note

Unsetting a field with a default value will revert back to the default.

`unique` (Default: False)

When True, no documents in the collection will have the same value for this field.

`unique_with` (Default: None)

A field name (or list of field names) that when taken together with this field, will not have two documents in the collection with the same value.

`primary_key` (Default: False)

When True, use this field as a primary key for the collection. *DictField* and *EmbeddedDocuments* both support being the primary key for a document.

Note

If set, this field is also accessible through the *pk* field.

`choices` (Default: None)

An iterable (e.g. list, tuple or set) of choices to which the value of this field should be limited.

Can either be nested tuples of value (stored in mongo) and a human readable key

```
SIZE = (('S', 'Small'),
        ('M', 'Medium'),
        ('L', 'Large'),
        ('XL', 'Extra Large'),
        ('XXL', 'Extra Extra Large'))


class Shirt(Document):
    size = StringField(max_length=3, choices=SIZE)
```

Or a flat iterable just containing values

```
SIZE = ('S', 'M', 'L', 'XL', 'XXL')

class Shirt(Document):
    size = StringField(max_length=3, choices=SIZE)
```

`validation` (Optional)

A callable to validate the value of the field. The callable takes the value as parameter and should raise a ValidationError if validation fails

e.g

```
def _not_empty(val):
    if not val:
        raise ValidationError('value can not be empty')

class Person(Document):
    name = StringField(validation=_not_empty)
```

`**kwargs` (Optional)

You can supply additional metadata as arbitrary additional keyword arguments. You can not override existing attributes, however. Common choices include *help_text* and *verbose_name*, commonly used by form and widget libraries.

2.3.3.2. List fields

MongoDB allows storing lists of items. To add a list of items to a `Document`, use the `ListField` field type. `ListField` takes another field object as its first argument, which specifies which type elements may be stored within the list:

```
class Page(Document):
    tags = ListField(StringField(max_length=50))
```

2.3.3.3. Embedded documents

MongoDB has the ability to embed documents within other documents. Schemata may be defined for these embedded documents, just as they may be for regular documents. To create an embedded document, just define a document as usual, but inherit from `EmbeddedDocument` rather than `Document`:

```
class Comment(EmbeddedDocument):
    content = StringField()
```

To embed the document within another document, use the `EmbeddedDocumentField` field type, providing the embedded document class as the first argument:

```
class Page(Document):
    comments = ListField(EmbeddedDocumentField(Comment))

comment1 = Comment(content='Good work!')
comment2 = Comment(content='Nice article!')
page = Page(comments=[comment1, comment2])
```

Embedded documents can also leverage the flexibility of dynamic-document-schemas: by inheriting `DynamicEmbeddedDocument`.

2.3.3.4. Dictionary Fields

Often, an embedded document may be used instead of a dictionary – generally embedded documents are recommended as dictionaries don't support validation or custom field types. However, sometimes you will not know the structure of what you want to store; in this situation a `DictField` is appropriate:

```
class SurveyResponse(Document):
    date = DateTimeField()
    user = ReferenceField(User)
    answers = DictField()

survey_response = SurveyResponse(date=datetime.utcnow(), user=request.user)
response_form = ResponseForm(request.POST)
survey_response.answers = response_form.cleaned_data()
survey_response.save()
```

Dictionaries can store complex data, other dictionaries, lists, references to other objects, so are the most flexible field type available.

2.3.3.5. Reference fields

References may be stored to other documents in the database using the `ReferenceField`. Pass in another document class as the first argument to the constructor, then simply assign document objects to the field:

```
class User(Document):
    name = StringField()

class Page(Document):
    content = StringField()
    author = ReferenceField(User)

john = User(name="John Smith")
john.save()

post = Page(content="Test Page")
post.author = john
post.save()
```

The `User` object is automatically turned into a reference behind the scenes, and dereferenced when the `Page` object is retrieved.

To add a `ReferenceField` that references the document being defined, use the string `'self'` in place of the document class as the argument to `ReferenceField`'s constructor. To reference a document that has not yet been defined, use the name of the undefined document as the constructor's argument:

```
class Employee(Document):
    name = StringField()
    boss = ReferenceField('self')
    profile_page = ReferenceField('ProfilePage')

class ProfilePage(Document):
    content = StringField()
```

2.3.3.5.1. Many to Many with ListFields

If you are implementing a many to many relationship via a list of references, then the references are stored as DBRefs and to query you need to pass an instance of the object to the query:

```
class User(Document):
    name = StringField()

class Page(Document):
    content = StringField()
    authors = ListField(ReferenceField(User))

bob = User(name="Bob Jones").save()
john = User(name="John Smith").save()

Page(content="Test Page", authors=[bob, john]).save()
Page(content="Another Page", authors=[john]).save()

# Find all pages Bob authored
Page.objects(authors__in=[bob])

# Find all pages that both Bob and John have authored
Page.objects(authors__all=[bob, john])

# Remove Bob from the authors for a page.
Page.objects(id='...').update_one(pull__authors=bob)

# Add John to the authors for a page.
Page.objects(id='...').update_one(push__authors=john)
```

2.3.3.5.2. Dealing with deletion of referred documents

By default, MongoDB doesn't check the integrity of your data, so deleting documents that other documents still hold references to will lead to consistency issues. Mongoengine's `ReferenceField` adds some functionality to safeguard against these kinds of database integrity problems, providing each reference with a delete rule specification. A delete rule is specified by supplying the `reverse_delete_rule` attributes on the `ReferenceField` definition, like this:

```
class ProfilePage(Document):
    employee = ReferenceField('Employee', reverse_delete_rule=mongoengine.CASCADE)
```

The declaration in this example means that when an `Employee` object is removed, the `ProfilePage` that references that employee is removed as well. If a whole batch of employees is removed, all profile pages that are linked are removed as well.

Its value can take any of the following constants:

`mongoengine.DO_NOTHING`

> This is the default and won't do anything. Deletes are fast, but may cause database inconsistency or dangling references.

`mongoengine.DENY`

> Deletion is denied if there still exist references to the object being deleted.

`mongoengine.NULLIFY`

Any object’s fields still referring to the object being deleted are set to None (using MongoDB’s “unset” operation), effectively nullifying the relationship.

`mongoengine.CASCADE`

Any object containing fields that are referring to the object being deleted are deleted first.

`mongoengine.PULL`

Removes the reference to the object (using MongoDB’s “pull” operation) from any object’s fields of `ListField` (`ReferenceField`).

Warning

A safety note on setting up these delete rules! Since the delete rules are not recorded on the database level by MongoDB itself, but instead at runtime, in-memory, by the MongoEngine module, it is of the upmost importance that the module that declares the relationship is loaded **BEFORE** the delete is invoked.

If, for example, the `Employee` object lives in the `payroll` app, and the `ProfilePage` in the `people` app, it is extremely important that the `people` app is loaded before any employee is removed, because otherwise, MongoEngine could never know this relationship exists.

In Django, be sure to put all apps that have such delete rule declarations in their `models.py` in the `INSTALLED_APPS` tuple.

2.3.3.5.3. Generic reference fields

A second kind of reference field also exists, `GenericReferenceField`. This allows you to reference any kind of `Document`, and hence doesn’t take a `Document` subclass as a constructor argument:

```
class Link(Document):
    url = StringField()

class Post(Document):
    title = StringField()

class Bookmark(Document):
    bookmark_object = GenericReferenceField()

link = Link(url='http://hmarr.com/mongoengine/')
link.save()

post = Post(title='Using MongoEngine')
post.save()

Bookmark(bookmark_object=link).save()
Bookmark(bookmark_object=post).save()
```

Note

Using `GenericReferenceField`s is slightly less efficient than the standard `ReferenceField`s, so if you will only be referencing one document type, prefer the standard `ReferenceField`.

2.3.3.6. Uniqueness constraints

MongoEngine allows you to specify that a field should be unique across a collection by providing `unique=True` to a `Field`'s constructor. If you try to save a document that has the same value for a unique field as a document that is already in the database, a `NotUniqueError` will be raised. You may also specify multi-field uniqueness constraints by using `unique_with`, which may be either a single field name, or a list or tuple of field names:

```
class User(Document):
    username = StringField(unique=True)
    first_name = StringField()
    last_name = StringField(unique_with='first_name')
```

2.3.4. Document collections

Document classes that inherit **directly** from `Document` will have their own **collection** in the database. The name of the collection is by default the name of the class converted to snake_case (e.g if your Document class is named *CompanyUser*, the corresponding collection would be *company_user*). If you need to change the name of the collection (e.g. to use MongoEngine with an existing database), then create a class dictionary attribute called `meta` on your document, and set `collection` to the name of the collection that you want your document class to use:

```
class Page(Document):
    title = StringField(max_length=200, required=True)
    meta = {'collection': 'cmsPage'}
```

2.3.4.1. Capped collections

A `Document` may use a **Capped Collection** by specifying `max_documents` and `max_size` in the `meta` dictionary. `max_documents` is the maximum number of documents that is allowed to be stored in the collection, and `max_size` is the maximum size of the collection in bytes. `max_size` is rounded up to the next multiple of 256 by MongoDB internally and mongoengine before. Use also a multiple of 256 to avoid confusions. If `max_size` is not specified and `max_documents` is, `max_size` defaults to 10485760 bytes (10MB). The following example shows a `Log` document that will be limited to 1000 entries and 2MB of disk space:

```
class Log(Document):
    ip_address = StringField()
    meta = {'max_documents': 1000, 'max_size': 2000000}
```

2.3.5. Indexes

You can specify indexes on collections to make querying faster. This is done by creating a list of index specifications called `indexes` in the `meta` dictionary, where an index specification may either be a single field name, a tuple containing multiple field names, or a dictionary containing a full index definition.

A direction may be specified on fields by prefixing the field name with a **+** (for ascending) or a **-** sign (for descending). Note that direction only matters on compound indexes. Text indexes may be specified by prefixing the field name with a **$**. Hashed indexes may be specified by prefixing the field name with a **#**:

```
class Page(Document):
    category = IntField()
    title = StringField()
    rating = StringField()
    created = DateTimeField()
    meta = {
        'indexes': [
            'title',   # single-field index
            '$title',  # text index
            '#title',  # hashed index
            ('title', '-rating'),  # compound index
            ('category', '_cls'),  # compound index
            {
                'fields': ['created'],
                'expireAfterSeconds': 3600  # ttl index
            }
        ]
    }
```

If a dictionary is passed then additional options become available. Valid options include, but are not limited to:

`fields` **(Default: None)**

The fields to index. Specified in the same format as described above.

`cls` **(Default: True)**

If you have polymorphic models that inherit and have `allow_inheritance` turned on, you can configure whether the index should have the `_cls` field added automatically to the start of the index.

`sparse` **(Default: False)**

Whether the index should be sparse.

`unique` (Default: False)

Whether the index should be unique.

`expireAfterSeconds` (Optional)

Allows you to automatically expire data from a collection by setting the time in seconds to expire the a field.

`name` (Optional)

Allows you to specify a name for the index

`collation` (Optional)

Allows to create case insensitive indexes (MongoDB v3.4+ only)

Note

Additional options are forwarded as **kwargs to pymongo's create_index method. Inheritance adds extra fields indices see: Document inheritance.

2.3.5.1. Global index default options

There are a few top level defaults for all indexes that can be set:

```python
class Page(Document):
    title = StringField()
    rating = StringField()
    meta = {
        'index_opts': {},
        'index_background': True,
        'index_cls': False,
        'auto_create_index': True,
        'auto_create_index_on_save': False,
    }
```

`index_opts` (Optional)

Set any default index options - see the full options list

`index_background` (Optional)

Set the default value for if an index should be indexed in the background

`index_cls` (Optional)

A way to turn off a specific index for _cls.

`auto_create_index` (Optional)

When this is True (default), MongoEngine will ensure that the correct indexes exist in MongoDB when the Document is first used. This can be disabled in systems where indexes are managed separately. Disabling this will improve performance.

`auto_create_index_on_save` (Optional)

> When this is True, MongoEngine will ensure that the correct indexes exist in MongoDB each time `save()` is run. Enabling this will degrade performance. The default is False. This option was added in version 0.25.

2.3.5.2. Compound Indexes and Indexing sub documents

Compound indexes can be created by adding the Embedded field or dictionary field name to the index definition.

Sometimes its more efficient to index parts of Embedded / dictionary fields, in this case use 'dot' notation to identify the value to index eg: *rank.title*

2.3.5.3. Geospatial indexes

The best geo index for mongodb is the new "2dsphere", which has an improved spherical model and provides better performance and more options when querying. The following fields will explicitly add a "2dsphere" index:

- `PointField`
- `LineStringField`
- `PolygonField`
- `MultiPointField`
- `MultiLineStringField`
- `MultiPolygonField`

As "2dsphere" indexes can be part of a compound index, you may not want the automatic index but would prefer a compound index. In this example we turn off auto indexing and explicitly declare a compound index on `location` and `datetime`:

```
class Log(Document):
    location = PointField(auto_index=False)
    datetime = DateTimeField()

    meta = {
        'indexes': [[("location", "2dsphere"), ("datetime", 1)]]
    }
```

2.3.5.3.1. Pre MongoDB 2.4 Geo

Note

For MongoDB < 2.4 this is still current, however the new 2dsphere index is a big improvement over the previous 2D model - so upgrading is advised.

Geospatial indexes will be automatically created for all `GeoPointField`s

It is also possible to explicitly define geospatial indexes. This is useful if you need to define a geospatial index on a subfield of a `DictField` or a custom field that contains a point. To create a geospatial index you must prefix the field with the * sign.

```
class Place(Document):
    location = DictField()
    meta = {
        'indexes': [
            '*location.point',
        ],
    }
```

2.3.5.4. Time To Live (TTL) indexes

A special index type that allows you to automatically expire data from a collection after a given period. See the official ttl documentation for more information. A common usecase might be session data:

```
class Session(Document):
    created = DateTimeField(default=datetime.utcnow)
    meta = {
        'indexes': [
            {'fields': ['created'], 'expireAfterSeconds': 3600}
        ]
    }
```

Warning

TTL indexes happen on the MongoDB server and not in the application code, therefore no signals will be fired on document deletion. If you need signals to be fired on deletion, then you must handle the deletion of Documents in your application code.

2.3.5.5. Comparing Indexes

Use `mongoengine.Document.compare_indexes()` to compare actual indexes in the database to those that your document definitions define. This is useful for maintenance purposes and ensuring you have the correct indexes for your schema.

2.3.6. Ordering

A default ordering can be specified for your `QuerySet` using the `ordering` attribute of `meta`. Ordering will be applied when the `QuerySet` is created, and can be overridden by subsequent calls to `order_by()`.

```
from datetime import datetime

class BlogPost(Document):
    title = StringField()
    published_date = DateTimeField()

    meta = {
        'ordering': ['-published_date']
    }

blog_post_1 = BlogPost(title="Blog Post #1")
blog_post_1.published_date = datetime(2010, 1, 5, 0, 0 ,0)

blog_post_2 = BlogPost(title="Blog Post #2")
blog_post_2.published_date = datetime(2010, 1, 6, 0, 0 ,0)

blog_post_3 = BlogPost(title="Blog Post #3")
blog_post_3.published_date = datetime(2010, 1, 7, 0, 0 ,0)

blog_post_1.save()
blog_post_2.save()
blog_post_3.save()

# get the "first" BlogPost using default ordering
# from BlogPost.meta.ordering
latest_post = BlogPost.objects.first()
assert latest_post.title == "Blog Post #3"

# override default ordering, order BlogPosts by "published_date"
first_post = BlogPost.objects.order_by("+published_date").first()
assert first_post.title == "Blog Post #1"
```

2.3.7. Shard keys

If your collection is sharded by multiple keys, then you can improve shard routing (and thus the performance of your application) by specifying the shard key, using the `shard_key` attribute of `meta`. The shard key should be defined as a tuple.

This ensures that the full shard key is sent with the query when calling methods such as `save()`, `update()`, `modify()`, or `delete()` on an existing `Document` instance:

```
class LogEntry(Document):
    machine = StringField()
    app = StringField()
    timestamp = DateTimeField()
    data = StringField()

    meta = {
        'shard_key': ('machine', 'timestamp'),
        'indexes': ('machine', 'timestamp'),
    }
```

2.3.8. Document inheritance

To create a specialised type of a `Document` you have defined, you may subclass it and add any extra fields or methods you may need. As this new class is not a direct subclass of `Document`, it will not be stored in its own collection; it will use the same collection as its superclass uses. This allows for more convenient and efficient retrieval of related documents – all you need do is set `allow_inheritance` to True in the `meta` data for a document.:

```
# Stored in a collection named 'page'
class Page(Document):
    title = StringField(max_length=200, required=True)

    meta = {'allow_inheritance': True}

# Also stored in the collection named 'page'
class DatedPage(Page):
    date = DateTimeField()
```

Note

From 0.8 onwards `allow_inheritance` defaults to False, meaning you must set it to True to use inheritance.

Setting `allow_inheritance` to True should also be used in `EmbeddedDocument` class in case you need to subclass it

When it comes to querying using `objects()`, querying *Page.objects()* will query both *Page* and *DatedPage* whereas querying *DatedPage* will only query the *DatedPage* documents. Behind the scenes, MongoEngine deals with inheritance by adding a `_cls` attribute that contains the class name in every documents. When a document is loaded, MongoEngine checks it's `_cls` attribute and use that class to construct the instance.:

```
Page(title='a funky title').save()
DatedPage(title='another title', date=datetime.utcnow()).save()

print(Page.objects().count())         # 2
print(DatedPage.objects().count())    # 1

# print documents in their native form
# we remove 'id' to avoid polluting the output with unnecessary detail
qs = Page.objects.exclude('id').as_pymongo()
print(list(qs))
# [
#   {'_cls': u 'Page', 'title': 'a funky title'},
#   {'_cls': u 'Page.DatedPage', 'title': u 'another title', 'date':
datetime.datetime(2019, 12, 13, 20, 16, 59, 993000)}
# ]
```

2.3.8.1. Working with existing data

As MongoEngine no longer defaults to needing `_cls`, you can quickly and easily get working with existing data. Just define the document to match the expected schema in your database

```
# Will work with data in an existing collection named 'cmsPage'
class Page(Document):
    title = StringField(max_length=200, required=True)
    meta = {
        'collection': 'cmsPage'
    }
```

If you have wildly varying schemas then using a `DynamicDocument` might be more appropriate, instead of defining all possible field types.

If you use `Document` and the database contains data that isn't defined then that data will be stored in the *document._data* dictionary.

2.3.9. Abstract classes

If you want to add some extra functionality to a group of Document classes but you don't need or want the overhead of inheritance you can use the `abstract` attribute of `meta`. This won't turn on Document inheritance but will allow you to keep your code DRY:

```
class BaseDocument(Document):
    meta = {
        'abstract': True,
    }
    def check_permissions(self):
        ...

class User(BaseDocument):
   ...
```

Now the User class will have access to the inherited *check_permissions* method and won't store any of the extra _*cls* information.