2.10. Documents migration

The structure of your documents and their associated mongoengine schemas are likely to change over the lifetime of an application. This section provides guidance and recommendations on how to deal with migrations.

Due to the very flexible nature of mongodb, migrations of models aren't trivial and for people that know about *alembic* for *sqlalchemy*, there is unfortunately no equivalent library that will manage the migration in an automatic fashion for mongoengine.

2.10.1. Example 1: Addition of a field

Let's start by taking a simple example of a model change and review the different option you have to deal with the migration.

Let's assume we start with the following schema and save an instance:

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

User(name="John Doe").save()

# print the objects as they exist in mongodb
print(User.objects().as_pymongo())    # [{u'_id': ObjectId('5d06b9c3d7c1f18db3e7c874'),
u'name': u'John Doe'}]
```

On the next version of your application, let's now assume that a new field *enabled* gets added to the existing `User` model with a *default=True*. Thus you simply update the `User` class to the following:

```
class User(Document):
    name = StringField(required=True)
    enabled = BooleanField(default=True)
```

Without applying any migration, we now reload an object from the database into the `User` class and checks its *enabled* attribute:

```
assert User.objects.count() == 1
user = User.objects().first()
assert user.enabled is True
assert User.objects(enabled=True).count() == 0    # uh?
assert User.objects(enabled=False).count() == 0   # uh?

# this is consistent with what we have in the database
# in fact, 'enabled' does not exist
print(User.objects().as_pymongo().first())    # {u'_id':
ObjectId('5d06b9c3d7c1f18db3e7c874'), u'name': u'John'}
assert User.objects(enabled=None).count() == 1
```

As you can see, even if the document wasn't updated, mongoengine applies the default value seamlessly when it loads the pymongo dict into a `User` instance. At first sight it looks like you don't need to migrate the existing documents when adding new fields but this actually leads to inconsistencies when it comes to querying.

In fact, when querying, mongoengine isn't trying to account for the default value of the new field and so if you don't actually migrate the existing documents, you are taking a risk that querying/updating will be missing relevant record.

When adding fields/modifying default values, you can use any of the following to do the migration as a standalone script:

```
# Use mongoengine to set a default value for a given field
User.objects().update(enabled=True)
# or use pymongo
user_coll = User._get_collection()
user_coll.update_many({}, {'$set': {'enabled': True}})
```

2.10.2. Example 2: Inheritance change

Let's consider the following example:

```
class Human(Document):
    name = StringField()
    meta = {"allow_inheritance": True}

class Jedi(Human):
    dark_side = BooleanField()
    light_saber_color = StringField()

Jedi(name="Darth Vader", dark_side=True, light_saber_color="red").save()
Jedi(name="Obi Wan Kenobi", dark_side=False, light_saber_color="blue").save()

assert Human.objects.count() == 2
assert Jedi.objects.count() == 2

# Let's check how these documents got stored in mongodb
print(Jedi.objects.as_pymongo())
# [
#   {'_id': ObjectId('5fac4aaaf61d7fb06046e0f9'), '_cls': 'Human.Jedi', 'name': 'Darth
Vader', 'dark_side': True, 'light_saber_color': 'red'},
#   {'_id': ObjectId('5fac4ac4f61d7fb06046e0fa'), '_cls': 'Human.Jedi', 'name': 'Obi
Wan Kenobi', 'dark_side': False, 'light_saber_color': 'blue'}
# ]
```

As you can observe, when you use inheritance, MongoEngine stores a field named '_cls' behind the scene to keep track of the Document class.

Let's now take the scenario that you want to refactor the inheritance schema and: - Have the Jedi's with dark_side=True/False become GoodJedi's/DarkSith - get rid of the 'dark_side' field

move to the following schemas:

```
# unchanged
class Human(Document):
    name = StringField()
    meta = {"allow_inheritance": True}

# attribute 'dark_side' removed
class GoodJedi(Human):
    light_saber_color = StringField()

# new class
class BadSith(Human):
    light_saber_color = StringField()
```

MongoEngine doesn't know about the change or how to map them with the existing data so if you don't apply any migration, you will observe a strange behavior, as if the collection was suddenly empty.

```
# As a reminder, the documents that we inserted
# have the _cls field = 'Human.Jedi'

# Following has no match
# because the query that is used behind the scene is
# filtering on {'_cls': 'Human.GoodJedi'}
assert GoodJedi.objects().count() == 0

# Following has also no match
# because it is filtering on {'_cls': {'$in': ('Human', 'Human.GoodJedi',
'Human.BadSith')}}
# which has no match
assert Human.objects.count() == 0
assert Human.objects.first() is None

# If we bypass MongoEngine and make use of underlying driver (PyMongo)
# we can see that the documents are there
humans_coll = Human._get_collection()
assert humans_coll.count_documents({}) == 2
# print first document
print(humans_coll.find_one())
# {'_id': ObjectId('5fac4aaaf61d7fb06046e0f9'), '_cls': 'Human.Jedi', 'name': 'Darth
Vader', 'dark_side': True, 'light_saber_color': 'red'}
```

As you can see, first obvious problem is that we need to modify '_cls' values based on existing values of 'dark_side' documents.

```
humans_coll = Human._get_collection()
old_class = 'Human.Jedi'
good_jedi_class = 'Human.GoodJedi'
bad_sith_class = 'Human.BadSith'
humans_coll.update_many({'_cls': old_class, 'dark_side': False}, {'$set': {'_cls':
good_jedi_class}})
humans_coll.update_many({'_cls': old_class, 'dark_side': True}, {'$set': {'_cls':
bad_sith_class}})
```

Let's now check if querying improved in MongoEngine:

```
assert GoodJedi.objects().count() == 1  # Hoorah!
assert BadSith.objects().count() == 1   # Hoorah!
assert Human.objects.count() == 2       # Hoorah!

# let's now check that documents load correctly
jedi = GoodJedi.objects().first()
# raises FieldDoesNotExist: The fields "{'dark_side'}" do not exist on the document
"Human.GoodJedi"
```

In fact we only took care of renaming the _cls values but we havn't removed the 'dark_side' fields which does not exist anymore on the GoodJedi's and BadSith's models. Let's remove the field from the collections:

```
humans_coll = Human._get_collection()
humans_coll.update_many({}, {'$unset': {'dark_side': 1}})
```

Note

We did this migration in 2 different steps for the sake of example but it could have been combined with the migration of the _cls fields:

```
humans_coll.update_many(
    {'_cls': old_class, 'dark_side': False},
    {
        '$set': {'_cls': good_jedi_class},
        '$unset': {'dark_side': 1}
    }
)
```

And verify that the documents now load correctly:

```
jedi = GoodJedi.objects().first()
assert jedi.name == "Obi Wan Kenobi"

sith = BadSith.objects().first()
assert sith.name == "Darth Vader"
```

An other way of dealing with this migration is to iterate over the documents and update/replace them one by one. This is way slower but it is often useful for complex migrations of Document models.

```
for doc in humans_coll.find():
    if doc['_cls'] == 'Human.Jedi':
        doc['_cls'] =  'Human.BadSith' if doc['dark_side'] else 'Human.GoodJedi'
        doc.pop('dark_side')
        humans_coll.replace_one({'_id': doc['_id']}, doc)
```

Warning

Be aware of this flaw if you modify documents while iterating

2.10.3. Example 4: Index removal

If you remove an index from your Document class, or remove an indexed Field from your Document class, you'll need to manually drop the corresponding index. MongoEngine will not do that for you.

The way to deal with this case is to identify the name of the index to drop with *index_information()*, and then drop it with *drop_index()*

Let's for instance assume that you start with the following Document class

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

    meta = {"indexes": ["name"]}

User(name="John Doe").save()
```

As soon as you start interacting with the Document collection (when *.save()* is called in this case), it would create the following indexes:

```
print(User._get_collection().index_information())
# {
#  '_id_': {'key': [('_id', 1)], 'v': 2},
#  'name_1': {'background': False, 'key': [('name', 1)], 'v': 2},
# }
```

Thus: '_id' which is the default index and 'name_1' which is our custom index. If you would remove the 'name' field or its index, you would have to call:

```
User._get_collection().drop_index('name_1')
```

Note

When adding new fields or new indexes, MongoEngine will take care of creating them (unless *auto_create_index* is disabled)

2.10.4. Recommendations

Write migration scripts whenever you do changes to the model schemas

Using `DynamicDocument` or `meta = {"strict": False}` may help to avoid some migrations or to have the 2 versions of your application to co-exist.

Write post-processing checks to verify that migrations script worked. See below

2.10.5. Post-processing checks

The following recipe can be used to sanity check a Document collection after you applied migration. It does not make any assumption on what was migrated, it will fetch 1000 objects randomly and run some quick checks on the documents to make sure the document looks OK. As it is, it will fail on the first occurrence of an error but this is something that can be adapted based on your needs.

```
def get_random_oids(collection, sample_size):
    pipeline = [{"$project": {'_id': 1}}, {"$sample": {"size": sample_size}}]
    return [s['_id'] for s in collection.aggregate(pipeline)]

def get_random_documents(DocCls, sample_size):
    doc_collection = DocCls._get_collection()
    random_oids = get_random_oids(doc_collection, sample_size)
    return DocCls.objects(id__in=random_oids)

def check_documents(DocCls, sample_size):
    for doc in get_random_documents(DocCls, sample_size):
        # general validation (types and values)
        doc.validate()

        # load all subfields,
        # this may trigger additional queries if you have ReferenceFields
        # so it may be slow
        for field in doc._fields:
            try:
                getattr(doc, field)
            except Exception:
                LOG.warning(f"Could not load field {field} in Document {doc.id}")
                raise

check_documents(Human, sample_size=1000)
```